
Cubicweb Documentation

Release 3.36.15

Logilab

March 28, 2023

GUIDES

1	A little history...	3
2	The Core Concepts of <i>CubicWeb</i>	5
2.1	Cubes	5
2.2	Instances	5
2.3	Data Repository	6
2.4	Web Engine	6
2.5	Schema (Data Model)	6
2.6	Registries and application objects	7
2.7	The RQL query language	8
2.8	Views	8
2.9	Hooks and operations	8
3	Tutorials	11
3.1	Building a simple blog with <i>CubicWeb</i>	11
3.2	Building a photo gallery with <i>CubicWeb</i>	27
3.3	Use Windmill with CubicWeb	59
3.4	Writing text reports with RestructuredText	62
3.5	Importing relational data into a CubicWeb instance	62
3.6	Create a Website from scratch with CubicWeb	71
4	Setup and Administration	93
4.1	Install a <i>CubicWeb</i> environment	93
4.2	Configure a <i>CubicWeb</i> environment	95
4.3	Deploy a <i>CubicWeb</i> application	100
4.4	cubicweb-ctl tool	102
4.5	Creation of your first instance	104
4.6	Configure an instance	105
4.7	User interface for web site configuration	108
4.8	Multiple sources of data	110
4.9	LDAP integration	110
4.10	RQL logs	112
5	Backend Development	113
5.1	Cubes	113
5.2	The Registry, selectors and application objects	118
5.3	Data model	125
5.4	Data as objects	143
5.5	Core APIs	154
5.6	Repository customization	155

5.7	Tests	172
5.8	Migration	184
5.9	Profiling and performance	187
5.10	Full Text Indexing in CubicWeb	188
5.11	Data Import	190
5.12	Debug Channels	194
5.13	API Reference	196
5.14	Source connections pooler	196
6	Web Frontend Development	199
6.1	Publisher	199
6.2	Controllers	201
6.3	The <i>Request</i> class (<i>cubicweb.web.request</i>)	202
6.4	RQL search bar	210
6.5	The View system	211
6.6	Configuring the user interface	248
6.7	Ajax	250
6.8	Javascript	252
6.9	CSS Stylesheet	257
6.10	Edition control	257
6.11	The facets system	285
6.12	Internationalization	292
6.13	The property mecanism	296
6.14	HTTP cache management	296
6.15	Locate resources	297
6.16	Static files handling	298
7	Pyramid	299
7.1	Quick start	299
7.2	The ‘pyramid’ command	300
7.3	Settings	301
7.4	Authentication	303
7.5	The pyramid debug toolbar	304
8	Additional Services	311
8.1	Undoing changes in CubicWeb	311
9	Appendixes	317
9.1	Frequently Asked Questions (FAQ)	317
9.2	Relation Query Language (RQL)	324
9.3	Introducing Mercurial	343
9.4	Installation dependencies	344
9.5	Javascript docstrings	345
10	Changelog	349
10.1	3.36.15 (2023-03-24)	349
10.2	3.36.14 (2023-03-02)	349
10.3	3.36.13 (2023-03-02)	349
10.4	3.36.12 (2023-01-17)	350
10.5	3.36.11 (2023-01-12)	350
10.6	3.36.10 (2022-11-15)	350
10.7	3.36.9 (2022-10-04)	350
10.8	3.36.8 (2022-09-22)	350
10.9	3.36.7 (2022-09-14)	351
10.10	3.36.6 (2022-08-30)	351

10.11	3.36.5 (2022-07-21)	351
10.12	3.36.4 (2022-07-13)	351
10.13	3.36.3 (2022-06-03)	351
10.14	3.36.2 (2022-06-01)	352
10.15	3.36.1 (2022-03-31)	352
10.16	3.36.0 (2022-03-14)	352
10.17	3.35.12 (2022-11-15)	353
10.18	3.35.11 (2022-10-04)	353
10.19	3.35.10 (2022-09-22)	353
10.20	3.35.9 (2022-09-14)	353
10.21	3.35.8 (2022-08-30)	353
10.22	3.35.7 (2022-07-21)	354
10.23	3.35.6 (2022-07-13)	354
10.24	3.35.5 (2022-07-13)	354
10.25	3.35.4 (2022-06-03)	354
10.26	3.35.3 (2022-06-01)	355
10.27	3.35.2 (2022-03-31)	355
10.28	3.35.1 (2022-03-09)	355
10.29	3.35 (2022-02-02)	355
10.30	3.34.3 (2022-03-31)	356
10.31	3.34.2 (2022-03-09)	356
10.32	3.34.1 (2021-12-01)	357
10.33	3.34.0 (2021-11-23)	357
10.34	3.33.13 (2022-03-09)	358
10.35	3.33.12 (2021-12-01)	358
10.36	3.33.11 (2021-11-17)	359
10.37	3.33.10 (2021-11-17)	359
10.38	3.33.9 (2021-11-08)	359
10.39	3.33.8 (2021-11-02)	359
10.40	3.33.7 (2021-10-12)	359
10.41	3.33.6 (2021-10-04)	360
10.42	3.33.5 (2021-09-29)	360
10.43	3.33.4 (2021-09-24)	360
10.44	3.33.3 (2021-09-14)	360
10.45	3.33.2 (2021-09-02)	360
10.46	3.33.1 (2021-08-31)	361
10.47	3.33.0 (2021-08-03)	361
10.48	3.32.14 (2021-12-01)	362
10.49	3.32.13 (2021-11-17)	362
10.50	3.32.12 (2021-11-17)	363
10.51	3.32.11 (2021-11-08)	363
10.52	3.32.10 (2021-11-02)	363
10.53	3.32.9 (2021-10-12)	363
10.54	3.32.8 (2021-10-04)	363
10.55	3.32.7 (2021-09-29)	364
10.56	3.32.6 (2021-09-24)	364
10.57	3.32.5 (2021-09-14)	364
10.58	3.32.4 (2021-09-02)	364
10.59	3.32.3 (2021-08-31)	364
10.60	3.32.2 (2021-07-30)	365
10.61	3.32.1 (2021-07-23)	365
10.62	3.32.0 (2021-07-13)	365
10.63	3.31.9 (2021-11-17)	368
10.64	3.31.8 (2021-11-17)	368

10.65	3.31.7 (2021-11-02)	368
10.66	3.31.6 (2021-09-28)	368
10.67	3.31.5 (2021-09-24)	368
10.68	3.31.4 (2021-09-14)	369
10.69	3.31.3 (2021-07-23)	369
10.70	3.31.2 (2021-07-19)	369
10.71	3.31.1 (2021-05-18)	369
10.72	3.31 (2021-05-04)	369
10.73	3.30.1 (2021-07-23)	370
10.74	3.30.0 (2021-03-16)	371
10.75	3.29.6 (2021-10-07)	373
10.76	3.29	373
10.77	3.28.2	374
10.78	3.28.1	374
10.79	3.28	375
10.80	3.27 (31 January 2020)	376
10.81	3.26 (1 February 2018)	378
10.82	3.25 (14 April 2017)	379
10.83	3.24 (2 November 2016)	380
10.84	3.23 (24 June 2016)	381
10.85	3.22 (4 January 2016)	382
10.86	3.21 (10 July 2015)	383
10.87	3.20 (06/01/2015)	385
10.88	3.19 (28/04/2015)	386
10.89	3.18 (10/01/2014)	388
10.90	3.17 (02/05/2013)	390
10.91	3.16 (25/01/2013)	391
10.92	3.15 (12/04/2012)	392
10.93	3.14 (09/11/2011)	394
11	API	397
11.1	cubicweb	397
11.2	cubicweb.appobject	401
11.3	cubicweb.cwvreg	402
11.4	logilab.common.registry	405
11.5	cubicweb.dataimport	410
11.6	cubicweb.predicates	411
11.7	cubicweb.pyramid	417
11.8	cubicweb.req	424
11.9	cubicweb.rset	427
11.10	cubicweb.web.views.urlpublishing	430
11.11	cubicweb.web.views.urlrewrite	431
11.12	cubicweb.web	432
12	CubicWeb - The Semantic Web is a construction game!	433
12.1	Main Features	433
12.2	First steps	433
12.3	Cubicweb core principle	434
12.4	Routing	434
12.5	Front development	435
12.6	Data model and management	435
12.7	Security	436
12.8	Migrate your schema	436
12.9	Cubicweb configuration files	436

12.10 Common Web application tools	436
12.11 Development	437
12.12 System administration	437
12.13 CubicWeb's ecosystem	438
12.14 How to contribute	438
Python Module Index	439
Index	441

This first part of the book offers different reading path to discover the *CubicWeb* framework, provides a tutorial to get a quick overview of its features and lists its key concepts.

A LITTLE HISTORY...

CubicWeb is a semantic web application framework that [Logilab](#) started developing in 2001 as an offspring of its [Narval](#) research project. *CubicWeb* is written in Python and includes a data server and a web engine.

Its data server publishes data federated from different sources like SQL databases, LDAP directories, [VCS](#) repositories or even from other *CubicWeb* data servers.

Its web engine was designed to let the final user control what content to select and how to display it. It allows one to browse the federated data sources and display the results with the rendering that best fits the context. This flexibility of the user interface gives back to the user some capabilities usually only accessible to application developers.

CubicWeb has been developed by [Logilab](#) and used in-house for many years before it was first installed for its clients in 2006 as version 2.

In 2008, *CubicWeb* version 3 became downloadable for free under the terms of the LGPL license. Its community is now steadily growing without hampering the fast-paced stream of changes thanks to the time and energy originally put in the design of the framework.

THE CORE CONCEPTS OF *CUBICWEB*

This section defines some terms and core concepts of the *CubicWeb* framework. To avoid confusion while reading this book, take time to go through the following definitions and use this section as a reference during your reading.

2.1 Cubes

A cube is a software component made of three parts:

- its data model (schema),
- its logic (entities) and
- its user interface (views).

A cube can use other cubes as building blocks and assemble them to provide a whole with richer functionalities than its parts. The cubes `cubicweb-blog` and `cubicweb-comment` could be used to make a cube named *myblog* with commentable blog entries.

The [CubicWeb.org Forge](#) offers a large number of cubes developed by the community and available under a free software license.

Note: The command `cubicweb-ctl list` displays the list of available cubes.

2.2 Instances

An instance is a runnable application installed on a computer and based on one or more cubes.

The instance directory contains the configuration files. Several instances can be created and based on the same cube. For example, several software forges can be set up on one computer system based on the `cubicweb-forge` cube.

The command `cubicweb-ctl list` also displays the list of instances installed on your system.

Note: The term application is used to refer to “something that should do something as a whole”, eg more like a project and so can refer to an instance or to a cube, depending on the context. This book will try to use *application*, *cube* and *instance* as appropriate.

2.3 Data Repository

The data repository¹ encapsulates and groups an access to one or more data sources (including SQL databases, LDAP repositories, other *CubicWeb* instance repositories, filesystems, Google AppEngine's DataStore, etc).

All interactions with the repository are done using the *Relation Query Language (RQL syntax)*. The repository federates the data sources and hides them from the querier, which does not realize when a query spans several data sources and requires running sub-queries and merges to complete.

Application logic can be mapped to data events happening within the repository, like creation of entities, deletion of relations, etc. This is used for example to send email notifications when the state of an object changes. See *Hooks and operations* below.

2.4 Web Engine

The web engine replies to http requests and runs the user interface.

By default the web engine provides a **CRUD** user interface based on the data model of the instance. Entities can be created, displayed, updated and deleted. As the default user interface is not very fancy, it is usually necessary to develop your own.

2.5 Schema (Data Model)

The data model of a cube is described as an entity-relationship schema using a comprehensive language made of Python classes imported from the *yams* library.

An *entity type* defines a sequence of attributes. Attributes may be of the following types: *String*, *Int*, *Float*, *Boolean*, *Date*, *Time*, *Datetime*, *Interval*, *Password*, *Bytes*, *RichString*.

A *relation type* is used to define an oriented binary relation between entity types. The left-hand part of a relation is named the *subject* and the right-hand part is named the *object*.

A *relation definition* is a triple (*subject entity type*, *relation type*, *object entity type*) associated with a set of properties such as cardinality, constraints, etc.

Permissions can be set on entity types or relation definition to control who will be able to create, read, update or delete entities and relations. Permissions are granted to groups (to which users may belong) or using rql expressions (if the rql expression returns some results, the permission is granted).

Some meta-data necessary to the system are added to the data model. That includes entities like users and groups, the entities used to store the data model itself and attributes like unique identifier, creation date, creator, etc.

When you create a new *CubicWeb* instance, the schema is stored in the database. When the cubes the instance is based on evolve, they may change their data model and provide migration scripts that will be executed when the administrator will run the upgrade process for the instance.

¹ not to be confused with a Mercurial repository or a Debian repository.

2.6 Registries and application objects

2.6.1 Application objects

Besides a few core functionalities, almost every feature of the framework is achieved by dynamic objects (*application objects* or *appobjects*) stored in a two-levels registry. Each object is affected to a registry with an identifier in this registry. You may have more than one object sharing an identifier in the same registry:

object's `__registry__` : object's `__regid__` : [list of app objects]

In other words, the *registry* contains several (sub-)registries which hold a list of appobjects associated to an identifier.

The base class of appobjects is `cubicweb.appobject.AppObject`.

2.6.2 Selectors

At runtime, appobjects can be selected in a registry according to some contextual information. Selection is done by comparing the *score* returned by each appobject's *selector*.

The better the object fits the context, the higher the score. Scores are the glue that ties appobjects to the data model. Using them appropriately is an essential part of the construction of well behaved cubes.

CubicWeb provides a set of basic selectors that may be parametrized. Also, selectors can be combined with the `~` unary operator (negation) and the binary operators `&` and `|` (respectively 'and' and 'or') to build more complex selectors. Of course complex selectors may be combined too. Last but not least, you can write your own selectors.

2.6.3 The *registry*

At startup, the *registry* inspects a number of directories looking for compatible class definitions. After a recording process, the objects are assigned to registries and become available through the selection process.

In a cube, application object classes are looked in the following modules or packages:

- *entities*
- *views*
- *hooks*
- *subjects*

There are three common ways to look up some application object from a registry:

- get the most appropriate object by specifying an identifier and context objects. The object with the greatest score is selected. There should always be a single appobject with a greater score than others for a particular context.
- get all objects applying to a context by specifying a registry. A list of objects will be returned containing the object with the highest score (> 0) for each identifier in that registry.
- get the object within a particular registry/identifier. No selection process is involved: the registry will expect to find a single object in that cell.

2.7 The RQL query language

No need for a complicated ORM when you have a powerful data manipulation language.

All the persistent data in a *CubicWeb* instance is retrieved and modified using RQL (see [Introduction](#)).

This query language is inspired by SQL but is on a higher level in order to emphasize browsing relations.

2.7.1 Result set

Every request made (using RQL) to the data repository returns an object we call a Result Set. It enables easy use of the retrieved data, providing a translation layer between the backend's native datatypes and *CubicWeb* schema's EntityTypes.

Result sets provide access to the raw data, yielding either basic Python data types, or schema-defined high-level entities, in a straightforward way.

2.8 Views

CubicWeb is data driven

The view system is loosely coupled to data through the selection system explained above. Views are application objects with a dedicated interface to 'render' something, eg producing some html, text, xml, pdf, or whatsoever that can be displayed to a user.

Views actually are partitioned into different kind of objects such as *templates*, *boxes*, *components* and proper *views*, which are more high-level abstraction useful to build the user interface in an object oriented way.

2.9 Hooks and operations

CubicWeb provides an extensible data repository

The data model defined using Yams types allows to express the data model in a comfortable way. However several aspects of the data model can not be expressed there. For instance:

- managing computed attributes
- enforcing complicated business rules
- real-world side-effects linked to data events (email notification being a prime example)

The hook system is much like the triggers of an SQL database engine, except that:

- it is not limited to one specific SQL backend (every one of them having an idiomatic way to encode triggers), nor to SQL backends at all (think about LDAP or a Mercurial repository)
- it is well-coupled to the rest of the framework

Hooks are also application objects (in the *hooks* registry) and selected on events such as after/before add/update/delete on entities/relations, server startup or shutdown, etc.

Operations may be instantiated by hooks to do further processing at different steps of the transaction's commit / rollback, which usually can not be done safely at the hook execution time.

Hooks and operation are an essential building block of any moderately complicated cubicweb application.

Note: RQL queries executed in hooks and operations are *unsafe* by default, i.e. the read and write security is deactivated unless explicitly asked.

TUTORIALS

Here are a few tutorials with different difficulty levels.

Beginners will want to start with the blog building tutorial giving a short introduction to the basic concepts. Then the photo gallery construction tutorial highlights more advanced concepts such as unit tests, security settings and migration scripts.

The other tutorials cover specific topics you can learn about when you understand the basics.

3.1 Building a simple blog with *CubicWeb*

CubicWeb is a semantic web application framework which favors reuse and object-oriented designs.

This tutorial is designed to help you make your very first steps with *CubicWeb*. It will guide you through basic concepts such as:

- getting an application running by using existing components
- discovering the default user interface
- extending and customizing the look and feel of that application

More advanced concepts are covered in *Building a photo gallery with CubicWeb*.

3.1.1 Some vocabulary

CubicWeb comes with a few words of vocabulary that you should know to understand what we're talking about. To follow this tutorial, you should at least know that:

- a *cube* is a component that usually includes a model defining some data types and a set of views to display them. A cube can be built by assembling other cubes;
- an *instance* is a specific installation of one or more cubes and includes configuration files, a web server and a database.

Reading *The Core Concepts of CubicWeb* for more vocabulary will be required at some point.

Now, let's start the hot stuff!

Get a blog running in five minutes!

First choose and follow *the installation method* of your choice.

Once you have *CubicWeb* setup, install the **blog cube** using the following command:

```
pip install cubicweb-blog
```

Then you can create and initialize your blog instance:

```
cubicweb-ctl create blog myblog
```

Here the **blog** argument tells the command to use the blog cube as a base for your instance named **myblog**.

Note: If you get a permission error of the kind `OSError: [Errno 13] Permission denied: '/etc/cubicweb.d/myblog'`, read the *next section*.

This command will ask you a series of question. The first one is about the database engine to use (SQLite or PostgreSQL). For this tutorial, we will use SQLite as it is easier to setup and does not need a database server. In production environments, PostgreSQL is recommended as it offers better performances. More information on database configuration can be found *here*.

The command will also create a user used to manage your instance, for which you will be asked to give a name and password.

You can leave the remaining questions to their default by simply pressing **Enter**.

Note: If you get errors during installation such as:

```
while handling language es: [Errno 2] No such file or directory: 'msgcat': 'msgcat'  
while handling language en: [Errno 2] No such file or directory: 'msgcat': 'msgcat'  
while handling language fr: [Errno 2] No such file or directory: 'msgcat': 'msgcat'
```

This means you are missing the `gettext` dependency. To fix this, follow the instructions in the section *Installing Dependencies*. Then either restart the installation process or run `cubicweb-ctl i18ncubicweb && cubicweb-ctl i18ncube blog` after installation. More information in *Internationalization*.

Then you need to tell *CubicWeb* your instance is going to run on the localhost by editing `~/etc/cubicweb.d/myblog/all-in-one.conf`. In this file under the `[MAIN]` section, replace the line `#host=` by `host=localhost`.

Once this process is complete (including database initialisation), you can start your instance by using:

```
cubicweb-ctl pyramid -D myblog
```

The `-D` option activates the debugging mode. Removing it will launch the instance as a daemon in the background.

This is it, your blog is functional and running at <http://localhost:8080!>

About file system permissions

Unless you *installed from source*, the above commands will initialize your instance as a regular user in your home directory (under `~/etc/cubicweb.d/`). If you installed from source, your instance will be created in system directories and thus will require root privileges. To change this behavior, please have a look at the *Resource mode* section.

Instance parameters

If you would like to change database parameters such as the database host or the user name used to connect to the database, edit the sources file located in the `/etc/cubicweb.d/myblog` directory.

Then relaunch the database creation:

```
cubicweb-ctl db-create myblog
```

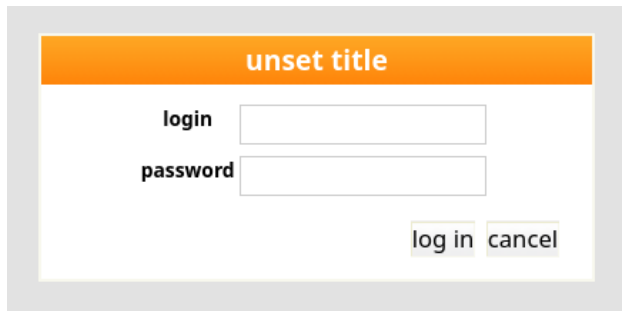
Other parameters, like web server or emails parameters, can be modified in the `/etc/cubicweb.d/myblog/all-in-one.conf` file (or `~/etc/cubicweb.d/myblog/all-in-one.conf` depending on your configuration.)

You'll have to restart the instance after modification in one of those files.

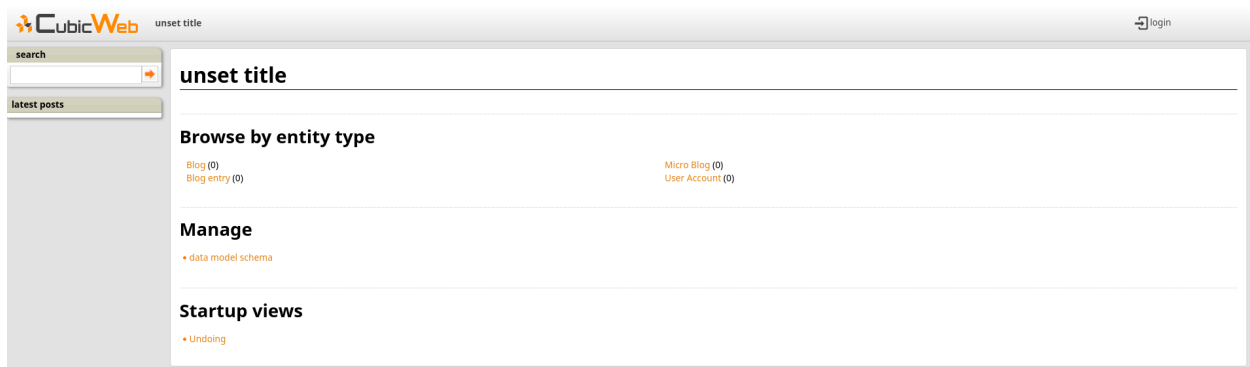
Discovering the web interface

You can now access your web instance to create blogs and post messages by visiting the URL <http://localhost:8080>.

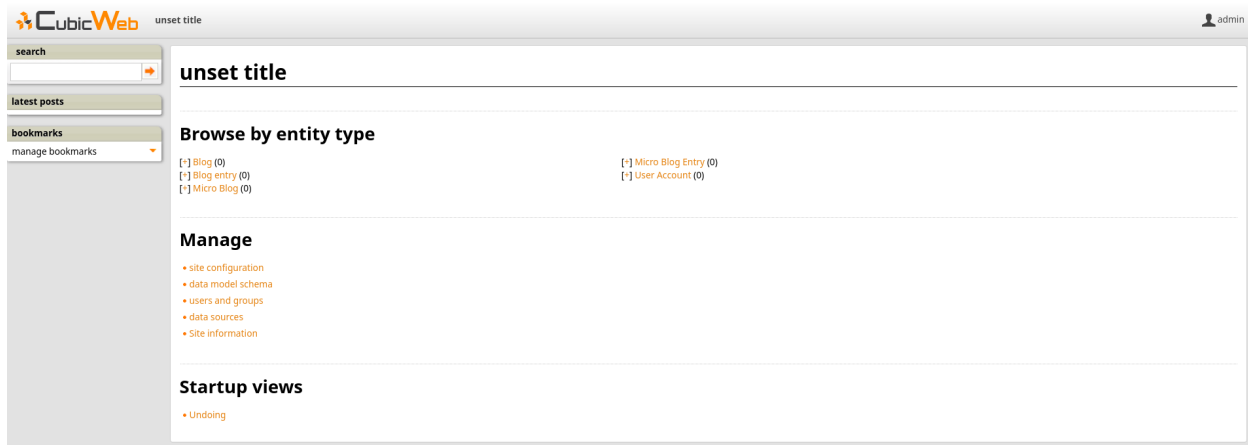
By default, anonymous access is disabled, so a login form will appear.



If you asked to allow anonymous access when initializing the instance, click on the 'login' link in the top right hand corner. To login, you need to use the admin account you specified at the time you initialized the database with `cubicweb-ctl create`.



Once authenticated, you can start playing with your instance. You will notice the index page has changed compared to the anonymous access view. There are more entries in the *Manage* section and some new [+] buttons have appeared next to the entities. These allow you to edit and add new entries in the database.



Note: If you find untranslated strings such as `blog.latest_blogs` in the sidebar:



This means you are missing the `gettext` dependency. To fix this, follow the instructions in the section [Installing Dependencies](#). Then either restart the installation process or run `cubicweb-ctl i18ncubicweb && cubicweb-ctl i18ncube blog` after installation. More information in [Internationalization](#).

Minimal configuration

Before creating entities, let's change the `unset title` string in the header. This string is set by a *CubicWeb* system properties and represents the site's title. To modify it, click on the `site configuration` link in the Manage section.

This will open a new page with different categories. You will find the site's title in the `ui` section. Simply set it to the desired value and click the 'button_ok' button.

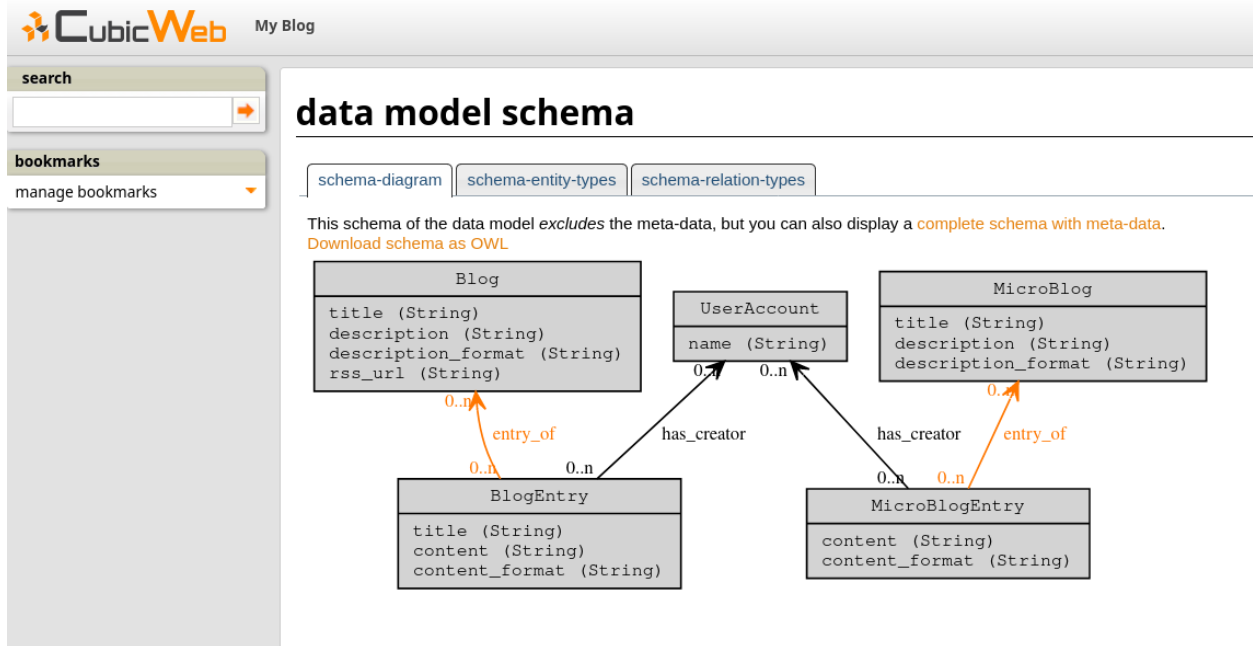
The screenshot shows the CubicWeb administration interface. On the left is a sidebar with a search bar and a 'bookmarks' section containing a 'manage bookmarks' link. The main content area is titled 'site configuration' and contains several expandable sections: 'navigation', 'ui', 'components', 'contextual components', and 'facets'. The 'ui' section is currently expanded, revealing a list of configuration options: 'date format' (with a text input field containing '%Y/%m/%d'), 'date and time format' (with a text input field containing '%Y/%m/%d %H:%M'), 'text format' (with a dropdown menu set to 'text/plain'), 'encoding' (with a text input field containing 'UTF-8'), 'float format' (with a text input field containing '%.3f'), 'language' (with a dropdown menu set to 'en'), 'main template' (with a text input field containing 'main-template'), 'site title' (with a text input field containing 'My Blog'), and 'time format' (with a text input field containing '%H:%M'). At the bottom of the 'ui' section is a green 'validate' button. The top of the page shows the CubicWeb logo, the text 'unset title', and a user profile icon labeled 'admin'.

You should see a `changes applied` message in green at the top of the section. You can now go back to the index page by clicking on the *CubicWeb* logo in the upper left-hand corner.

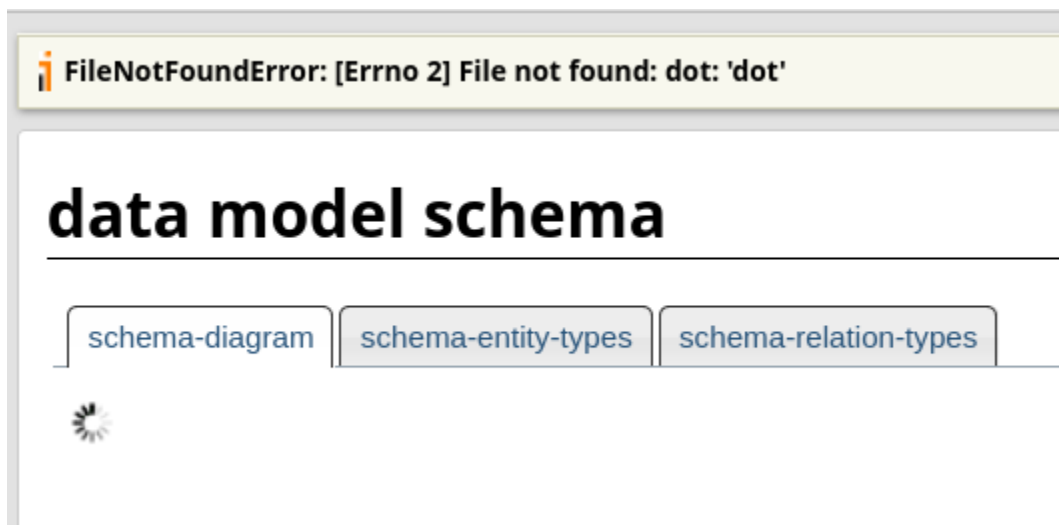
You will much likely still see `unset title` at this point. This is because by default the index page is cached for performance reasons. Force a refresh of the page (Ctrl-R in Firefox) and you should now see the title you entered.

Adding entities

The blog cube defines several entity types. For example, the `Blog` entity is a container for a `BlogEntry` (i.e. posts) on a particular topic. We can get a graphical view of the schema by clicking on the `data model schema` link in the `Manage` section of the index page:



Note: If you get the error `FileNotFoundException: [Errno 2] File not found: dot: 'dot'` when accessing the page, this means you are missing the package `graphviz`. To fix this, follow the instructions in the section [Installing Dependencies](#).



Notice that like most other things we will see in this tutorial, this schema is generated by the framework according to the application's model. In our case the model is defined by the `blog` cube.

Now let's create a few of those entities.

Adding a blog

Clicking on the `[+]` at the left of the `Blog` link on the index page will open an HTML form to create a new blog.

For instance, call this new blog `Tech-blog` and type in `everything about technology` as the description, then validate the form by clicking on `validate`. You will be redirected to the *primary* view of the newly created blog.

Adding a blog post

There are several ways to add a blog entry. The simplest is to click on the `add blog entry` link in the actions box on the left while viewing the blog you just created. You will then see a form to create a post, with a `blog entry` of field preset to the blog you are coming from. Enter a title, some content, click the `validate` button and you're done. You will be redirected to the blog's primary view, though you now see that it contains the blog post you have just created.

Notice how some new items appeared in the left column.

You can achieve the same result by clicking on the `[+]` at the left of the `Blog entry` link on the index page. Since there is no context information, the `blog entry` of selector will not be preset to a blog if you have more than one.

If you click on the `modify` link in the action box, you will be taken back to the form to edit the entity you just created. But the form will now have another section with a combo-box entitled `add relation` providing a generic way to edit relations. Choose the relation you want to add and a second combo box will appear where you can pick existing entities.

If there are too many of them, you will be offered to navigate to the target entity. This will open a new page and you will be taken back to your form once you have selected an entity.

This combo-box cannot appear until the entity is actually created, explaining why you could not see it at creation time using the first form. Another way to show this combo-box is to hit **apply** instead of **validate** to create the entity without closing the form.

About UI auto-adaptation

One of the things making *CubicWeb* different from other frameworks is its automatic user interface adapting itself according to the data being displayed. Let's see an example.

If you go back to the home page and click on the **Blog** link, you will be redirected to the blog's primary view as we have seen earlier. Now add another blog, go back to the index page, and click again on this link. You will see a very different view (namely the *list* view).

In the first case the framework chose to use the *primary* view since there was only one entity in the data to be displayed. Now that there are two entities, the *list* view is more appropriate and hence is being used.

There are various other places where *CubicWeb* adapts to display data in the best way, the main being provided by the view *selection* mechanism that will be detailed later.

Digging deeper

By following the principles explained above you should now be able to create new users for your application and to configure your instance. You will notice that the index page lists a lot of types we did not talk know about. Most are built-in types provided by the framework to make the whole system work. You may ignore them in a first time and discover them as time goes.

One thing that is worth playing with is the search box. It may be used in various ways, from simple full text search to advanced queries using the *RQL syntax*.

Customizing your application

Usually you won't get enough by assembling cubes out-of-the-box. You will want to customize them to get personal look and feel, add your own data model and so on. Or maybe start from scratch?

So let's get a bit deeper and start coding our own cube. In our case, we want to customize the blog we created to add more features to it.

Creating your own cube

Once your *CubicWeb* development environment is set up, you can create a new cube:

```
cubicweb-ctl newcube mycube
```

This will create a directory named `cubicweb-mycube` reflecting the structure described in *Standard structure for a cube*.

To install your new cube on the virtual environment created previously, run the following command in `cubicweb-mycube` directory:

```
pip install -e .
```

All *cubicweb-ctl* commands are described in details in *cubicweb-ctl tool*.

Cube metadata

The folder `cubicweb_mycube/` contains the actual code and metadata for your cube. In this folder, a simple set of metadata about your cube are stored in the `__pkginfo__.py` file. In our case, we want to extend the blog cube, so we have to tell that our cube depends on this cube by modifying the `__depends__` dictionary in that file:

```
__depends__ = {"cubicweb": ">= 3.35.0", "cubicweb-blog": None}
```

where `None` means we do not depend on a particular version of the cube.

Extending the data model

The data model or schema is the core of your *CubicWeb* application. It defines the type of content your application will handle. It is defined in the file `schema.py` of the cube.

Defining our model

Let's say we want a new entity type named *Community* with a name and a description. A *Community* will hold several blogs.

We can edit the `schema.py` as follows:

```
from yams.buildobjs import EntityType, RelationDefinition, String, RichString

class Community(EntityType):
    name = String(maxsize=50, required=True)
    description = RichString()
```

(continues on next page)

(continued from previous page)

```
class community_blog(RelationDefinition):
    subject = 'Community'
    object = 'Blog'
    cardinality = '*?'
    composite = 'subject'
```

The import from the yams package provides necessary classes to build the schema.

This file defines the following:

- a *Community* has a name and a description as attributes
 - the name is a string which is required and cannot be longer than 50 characters
 - the description is an unconstrained string and may contains rich content such as HTML or Restructured text.
- a *Community* may be linked to a *Blog* using the *community_blog* relation
 - * means a community may be linked from 0 to N blog, ? means a blog may be linked to 0 to 1 community. For completeness, you can also use + for 1 to N, and 1 for a single mandatory relation (e.g. one to one);
 - this is a composite relation where *Community* (e.g. the subject of the relation) is the composite. That means that if you delete a community, its blog will be deleted as well.

Of course, there are a lot of other data types and relations such as constraints, permissions, etc, that may be defined in the schema but those will not be covered in this tutorial.

Notice that our schema refers to the *Blog* entity type which is not defined here. But we know this type is available since we depend on the *blog* cube defining it.

Applying changes from the model into our instance

The problem is that we created an instance using the *blog* cube, not our *mycube* cube. If we do not do anything there is no way we'll see anything changing in the *myblog* instance.

As we do not have any really valuable data in the instance, an easy way would be to trash it and recreated it. First stop the running instance by pressing Ctrl-C in the terminal running the server in debug mode. Then run the following commands:

```
cubicweb-ctl delete myblog
cubicweb-ctl create mycube myblog
cubicweb-ctl pyramid -D myblog
```

Another way is to add our cube to the instance using the `cubicweb-ctl shell` facility. It is a python shell connected to the instance with some special commands available to manipulate it (the same as you'll have in migration scripts, which are not covered in this tutorial). In that case, we are interested in the `add_cube` command. First stop the instance by pressing Ctrl-C in the terminal running the server in debug mode and enter the shell using the following command:

```
cubicweb-ctl shell myblog
```

Then in the python shell, type the `add_cube` command:

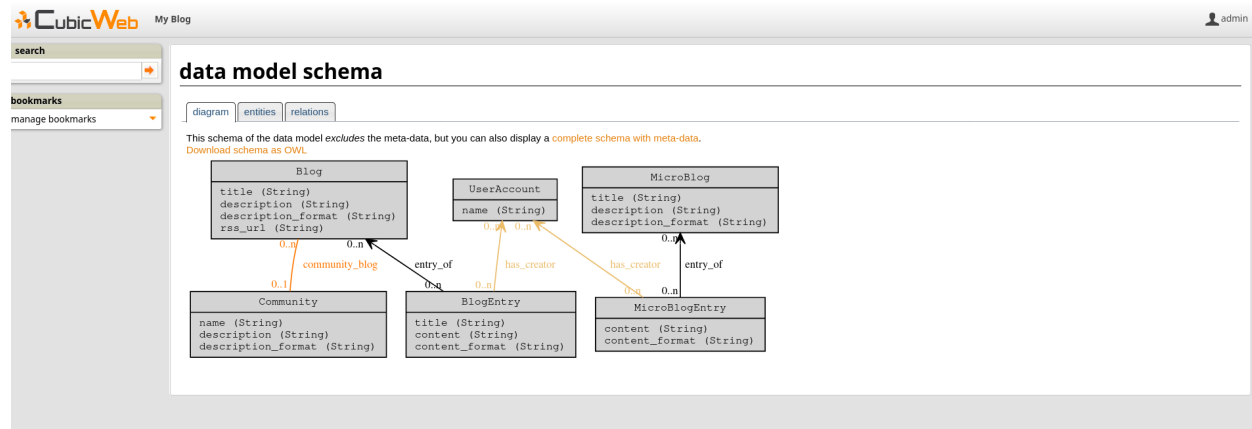
```
add_cube('mycube')
```

Press Ctrl-D to exit then restart your instance:

```
cubicweb-ctl pyramid -D myblog
```

The `add_cube` command is enough since it automatically updates our application to the cube's schema. There are plenty of other migration commands of a more finer grain. They are described in [Migration](#)

If you take another look at the schema on your instance, you will see that changes to the data model have actually been applied (meaning database schema updates and all necessary actions have been done).



If you follow the **Site information** link in the home page, you will also see that the instance is using blog and mycube cubes (sioc is a dependency of the blog cube).

versions configuration	
CubicWeb	3.35.0
blog	1.14.1
mycube	0.1.0
sioc	0.2.2

You can now add some communities and link them to a blog. You will see that the framework provides default views for this entity type (we have not yet defined any view for it!), and also that the blog primary view will show the community it is linked to if any. All this thanks to the model driven interface provided by the framework.

We will now see how to redefine each of them according to your needs and preferences.

Defining your views

CubicWeb provides a lot of standard views in the directory `cubicweb/web/views/`. We already talked about *primary* and *list* views, which are views applying to one or more entities.

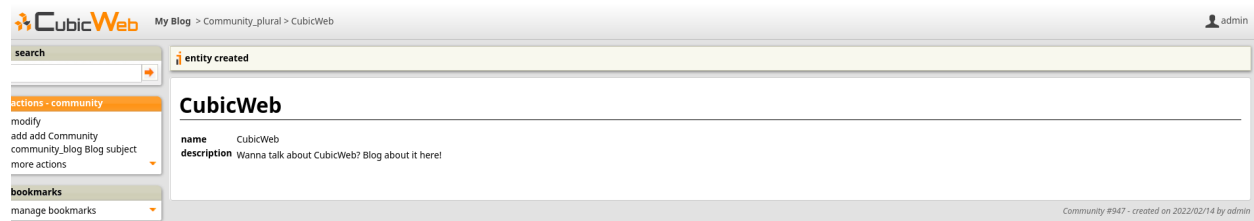
A view is defined by a python class which includes:

- an identifier: all objects used to build the user interface in *CubicWeb* are recorded in a registry and this identifier will be used as a key in that registry to store the view. There may be multiple views for the same identifier.
- a *selector*, which is a kind of filter telling how well a view suits to a particular context. When looking for a particular view (e.g. given an identifier), *CubicWeb* computes for each available view with that identifier a score which is returned by the selector. Then the view with the highest score is used. The standard library of predicates is in `cubicweb.predicates`.

A view has a set of methods inherited from the `cubicweb.view.View` class, though you do not usually derive directly from this class but from one of its more specific child class.

Last but not least, *CubicWeb* provides a set of default views accepting any kind of entities.

To illustrate this, we will create a community as we already have done for other entity types through the index page. You will get a screen similar to this:



Changing the layout of the application

The layout is the general organization of the pages in the website. Views generating the layout are sometimes referred to as *templates*. They are implemented by the framework in the module `cubicweb.web.views.basetemplates`. By overriding classes in this module, you can customize whatever part you wish of the default layout.

CubicWeb provides many other ways to customize the interface thanks to actions and components (which you can individually (de)activate, control their location, customize their look...) as well as “simple” CSS customization. You should first try to achieve your goal using such fine grained parametrization rather than overriding a whole template, which usually embeds customisation access points that you may lose in the process.

But for the sake of example, let’s say we want to change the generic page footer. We can simply add in the file `cubicweb_mycube/views.py` the code below:

```
from cubicweb.web.views import basetemplates

class MyHTMLPageFooter(basetemplates.HTMLPageFooter):

    def footer_content(self):
        self.w(u'This website has been created with <a href="http://cubicweb.org">
↪CubicWeb</a>.')

def registration_callback(vreg):
    vreg.register_all(globals().values(), __name__, (MyHTMLPageFooter,))
    vreg.register_and_replace(MyHTMLPageFooter, basetemplates.HTMLPageFooter)
```

- Our class inherits from the default page footer to ease getting things right, but this is not mandatory.
- When we want to write something to the output stream, we simply call `self.w`, which *must be passed a unicode string*.
- Since both `HTMLPageFooter` and `MyHTMLPageFooter` have the same selector, hence the same score the framework would not be able to choose which footer to use. In this case we want our footer to replace the default one, so we have to define a `registration_callback()` function to control object registration. The first instruction tells to register everything in the module but the `MyHTMLPageFooter` class, then the second to register it instead of `HTMLPageFooter`. Without this function, everything in the module is registered blindly.

Note: When a view is modified while running in debug mode, it is not required to restart the instance server. Save the Python file and reload the page in your web browser to view the changes.

You will now see this simple footer on every page of the website.

Primary view customization

The *primary* view (i.e. any view with the identifier set to *primary*) is the one used to display all the information about a single entity. The standard primary view is one of the most sophisticated views of all. It has several customisation points, but its power comes with *uicfg* allowing you to control it without having to subclass it.

However this is a bit off-topic for this first tutorial. Let's say we simply want a custom primary view for the *Community* entity type, using directly the view interface without trying to benefit from the default implementation (you should do that though if you're rewriting reusable cubes; everything is described in more details in [The Primary View](#)).

here is the code that we will put in the file `cubicweb_mycube/views.py` of our cube:

```
from cubicweb.predicates import is_instance
from cubicweb.web.views import primary

class CommunityPrimaryView(primary.PrimaryView):
    __select__ = is_instance('Community')

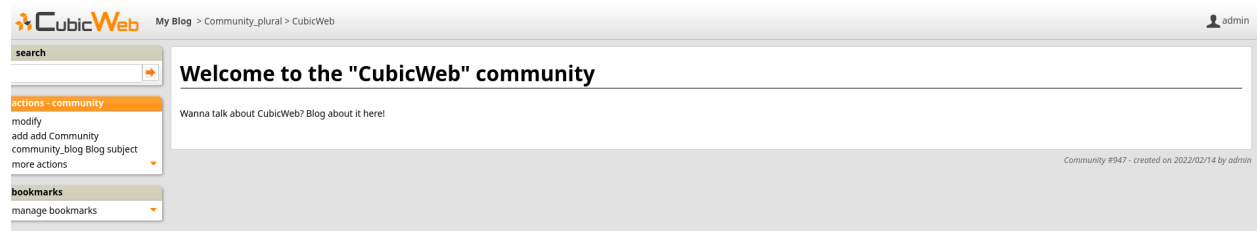
    def cell_call(self, row, col):
        entity = self.cw_rset.get_entity(row, col)
        self.w(u'<h1>Welcome to the "%s" community</h1>' % entity.printable_value('name
→'))

        if entity.description:
            self.w(u'<p>%s</p>' % entity.printable_value('description'))
```

What's going on here?

- Our class inherits from the default primary view, here mainly to get the correct view identifier, since we do not use any of its features.
- We set on it a selector telling that it only applies when trying to display some entity of the *Community* type. This is enough to get an higher score than the default view for entities of this type.
- A view that applies to an entity usually has to define the method `cell_call` as an entry point. This receives the arguments `row` and `col` telling to which entity in the result set the view is applied. We can then get this entity from the result set (`self.cw_rset`) by using the `get_entity` method.
- To ease thing, we access our entity's attribute to display using its `printable_value` method, which will handle formatting and escaping when necessary. As you can see, you can also access attributes by their name on the entity to get the raw value.

You can now reload the page of the community we just created and see the changes.



We have seen here a lot of thing you will have to deal with to write views in *CubicWeb*. The good news is that this is almost everything that is used to build higher level layers.

Note: As things get complicated and the volume of code in your cube increases, you can of course still split your views module into a python package with subpackages.

You can find more details about views and selectors in *Principles*.

Write entities to add logic in your data

CubicWeb provides an ORM (Object-Relational Mapper) to programmatically manipulate entities (just like the one we have fetched earlier by calling `get_entity` on a result set). By default, entity types are instances of the `AnyEntity` class, which holds a set of predefined methods as well as properties automatically generated for attributes/relations of the type it represents.

You can redefine each entity to provide additional methods or whatever you want to help you write your application. Customizing an entity requires that your entity:

- inherits from `cubicweb.entities.AnyEntity` or any subclass
- defines a `__regid__` linked to the corresponding data type of your schema

You may then want to add your own methods, override default implementation of some method, etc... To do so, write this code in `mycube/entities.py`:

```
from cubicweb.entities import AnyEntity, fetch_config

class Community(AnyEntity):
    """customized class for Community entities"""
    __regid__ = 'Community'

    fetch_attrs, cw_fetch_order = fetch_config(['name'])

    def dc_title(self):
        return self.name

    def display_cw_logo(self):
        return 'CubicWeb' in self.name
```

In this example:

- we used the `fetch_config()` convenience function to tell which attributes should be prefetched by the ORM when looking for some related entities of this type, and how they should be ordered
- we overrode the standard `dc_title()` method, used in various place in the interface to display the entity (though in this case the default implementation would have had the same result)

- we implemented here a method `display_cw_logo()` which tests if the community title contains *CubicWeb*. It can then be used when you are writing code involving *Community* entities in your views, hooks, etc. For instance, you can modify your previous views as follows:

```
class CommunityPrimaryView(primary.PrimaryView):
    __select__ = is_instance('Community')

    def cell_call(self, row, col):
        entity = self.cw_rset.get_entity(row, col)
        self.w(u'<h1>Welcome to the "%s" community</h1>' % entity.printable_value('name
→'))

        if entity.display_cw_logo():
            self.w(u'')

        if entity.description:
            self.w(u'<p>%s</p>' % entity.printable_value('description'))
```

Then each community whose description contains 'CW' is shown with the *CubicWeb* logo in front of it.

Note: As for view, you don't have to restart your instance when modifying some entity classes while your server is running in debug mode, the code will be automatically reloaded.

Extending the application by using more cubes!

One of the goals of the *CubicWeb* framework is to have truly reusable components. To do so they must behave nicely when plugged into the application and be easily customisable, from the data model to the user interface. Thanks to systems such as the selection mechanism and the choice to write views as python code, we can build our pages using true object oriented programming techniques to achieve this goal.

A library of standard cubes is available at the [CubicWeb Forge](#) to address a lot of common problems such as manipulating files, people, todos, etc. In our community blog case, we could be interested for instance in functionalities provided by the `comment` and `tag` cubes. `comment` provides threaded discussion functionalities and `tag` a simple tag mechanism to classify content. We will first modify our cube's `__pkginfo__.py` file to add those cubes as dependencies:

```
__depends__ = {'cubicweb': '>= 3.35.0',
              'cubicweb-blog': None,
              'cubicweb-comment': None,
              'cubicweb-tag': None}
```

Now we will simply tell on which entity types we want to activate the `comment` and `tag` cubes by adding respectively the `comments` and `tags` relations on them in our schema (`schema.py`).

```
class comments(RelationDefinition):
    subject = 'Comment'
    object = 'BlogEntry'
    cardinality = '1*'
    composite = 'object'

class tags(RelationDefinition):
```

(continues on next page)

(continued from previous page)

```
subject = 'Tag'
object = ('Community', 'BlogEntry')
```

In the above code we activated comments on BlogEntry entities and tags on both Community and BlogEntry. Various views from both comment and tag cubes will then be automatically displayed when one of those relations is supported.

Let's install the cubes and synchronize the data model as we've done earlier. So first install the cubes:

```
pip install cubicweb-comment cubicweb-tag
```

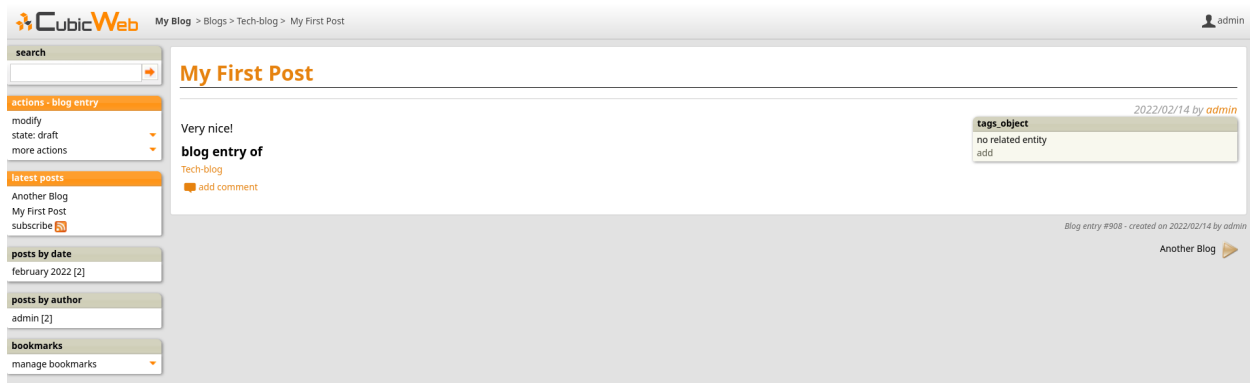
Stop the instance by pressing Ctrl-C in the terminal running the server in debug mode and enter the migration shell:

```
cubicweb-ctl shell myblog
```

Add the new cubes and exit with Ctrl-D:

```
add_cubes(('comment', 'tag'))
```

Then restart the instance with `cubicweb-ctl pyramid -D myblog` and open a blog entry:



As you can see, we now have a box displaying tags and a section proposing to add a comment and displaying existing one below the post. All this without changing anything in our views, thanks to the design of generic views provided by the framework. Though if we take a look at a community, we will not see the tags box! This is because by default this box tries to locate itself in the right column within the white frame, and this column is handled by the primary view we overrode. Let's change our view to make it more extensible, by keeping both our custom rendering but also extension points provided by the default implementation.

Add the following code in `cubicweb_mycube/views.py`:

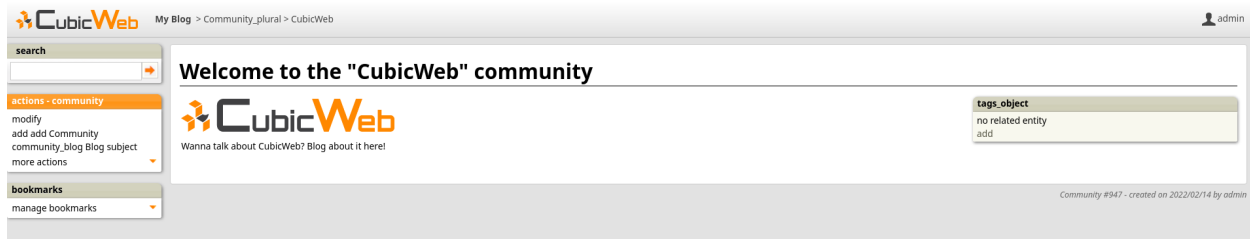
```
class CommunityPrimaryView(primary.PrimaryView):
    __select__ = is_instance('Community')

    def render_entity_title(self, entity):
        self.w(u'<h1>Welcome to the "%s" community</h1>' % entity.printable_value('name
↪'))

    def render_entity_attributes(self, entity):
        if entity.display_cw_logo():
            self.w(u'')

        if entity.description:
            self.w(u'<p>%s</p>' % entity.printable_value('description'))
```

By reloading the Community page, it will now appear properly:



You can control part of the interface independently from each others, piece by piece.

What's next?

In this tutorial we have seen you can build a web application in a few minutes simply by defining a data model. You get a working application which you can then customize without breaking your workflow. You can show results to customers right from the beginning to make the right decisions early in the process. This is important in agile development practices.

The next steps will be to discover hooks, security, data sources, digging deeper into view writing and interface customisation... Still a lot of fun stuff to discover! You will find more tutorials and howtos in the blog published on the CubicWeb.org website.

3.2 Building a photo gallery with *CubicWeb*

3.2.1 Desired features

- basically a photo gallery
- photo stored on the file system and displayed dynamically through a web interface
- navigation through folder (album), tags, geographical zone, people on the picture... using facets
- advanced security (not everyone can see everything). More on this later.

Cube creation and schema definition

Step 1: creating a virtual environment

Fisrt I need a python virtual environment with cubicweb:

```
python3 -m venv venv
source venv/bin/activate
pip install cubicweb
```

Step 2: creating a new cube for my web site

One note about my development environment: I wanted to use the packaged version of CubicWeb and cubes while keeping my cube in the current directory, let's say `~/src/cubes`:

```
cd ~/src/cubes
CW_MODE=user
```

I can now create the cube which will hold custom code for this web site using:

```
cubicweb-ctl newcube sytweb
```

Enter a short description and this will create your new cube in the `cubicweb-sytweb` folder.

Step 3: pick building blocks into existing cubes

Almost everything I want to handle in my web-site is somehow already modeled in existing cubes that I'll extend for my need. So I'll pick the following cubes:

- **folder**, containing the *Folder* entity type, which will be used as both 'album' and a way to map file system folders. Entities are added to a given folder using the *filed_under* relation.
- **file**, containing *File* entity type, gallery view, and a file system import utility.
- **person**, containing the *Person* entity type plus some basic views.
- **comment**, providing a full commenting system allowing one to comment entity types supporting the *comments* relation by adding a *Comment* entity.
- **tag**, providing a full tagging system as an easy and powerful way to classify entities supporting the *tags* relation by linking the to *Tag* entities. This will allows navigation into a large number of picture.

Ok, now I'll tell my cube requires all this by editing `cubicweb-sytweb/cubicweb_sytweb/__pkginfo__.py`:

```
__depends__ = {'cubicweb': '>= 3.32.7',
              'cubicweb-file': '>= 1.9.0',
              'cubicweb-folder': '>= 1.1.0',
              'cubicweb-person': '>= 1.2.0',
              'cubicweb-comment': '>= 1.2.0',
              'cubicweb-tag': '>= 1.2.0'
             }
```

Notice that you can express minimal version of the cube that should be used, *None* meaning whatever version available. All packages starting with 'cubicweb-' will be recognized as being cube, not bare python packages.

Now, I need to install all the dependencies:

```
cd cubicweb-sytweb
pip install -e .
pip install cubicweb
pip install psycopg2-binary # for postgresql
```

Step 4: glue everything together in my cube's schema

Put this code in `cubicweb-sytweb/cubicweb_sytweb/schema.py`:

```
from yams.buildobjs import RelationDefinition

class comments(RelationDefinition):
    subject = 'Comment'
    object = 'File'
    # a Comment can be on only one File
    # but a File can have several comments
    cardinality = '1*'
    composite = 'object'

class tags(RelationDefinition):
    subject = 'Tag'
    object = 'File'

class filed_under(RelationDefinition):
    subject = 'File'
    object = 'Folder'

class displayed_on(RelationDefinition):
    subject = 'Person'
    object = 'File'
```

This schema:

- allows to comment and tag on *File* entity type by adding the *comments* and *tags* relations. This should be all we've to do for this feature since the related cubes provide 'pluggable section' which are automatically displayed on the primary view of entity types supporting the relation.
- adds a *situated_in* relation definition so that image entities can be geolocalized.
- add a new relation *displayed_on* relation telling who can be seen on a picture.

This schema will probably have to evolve as time goes (for security handling at least), but since the possibility to let a schema evolve is one of CubicWeb's features (and goals), we won't worry about it for now and see that later when needed.

Step 5: creating the instance

Now that I have a schema, I want to create an instance. To do so using this new 'sytweb' cube, I run:

```
cubicweb-ctl create sytweb sytweb_instance
```

For simplicity you should use the sqlite database, it won't require configuration.

Don't forget to say "yes" to the question: *Allow anonymous access ? [y/N]*:

Hint: if you get an error while the database is initialized, you can avoid having to answer the questions again by running:

```
cubicweb-ctl db-create sytweb_instance
```

This will use your already configured instance and start directly from the create database step, thus skipping questions asked by the ‘create’ command.

Once the instance and database are fully initialized, run

```
cubicweb-ctl pyramid -D sytweb_instance
```

to start the instance, check you can connect on it, etc... then go on <http://localhost:8080> (or with another port if you’ve modified it)

Security, testing and migration

This part will cover various topics:

- configuring security
- migrating existing instance
- writing some unit tests

Here is the read security model I want:

- folders, files, images and comments should have one of the following visibility:
 - **public**, everyone can see it
 - **authenticated**, only authenticated users can see it
 - **restricted**, only a subset of authenticated users can see it
- managers (e.g. me) can see everything
- only authenticated users can see people
- everyone can see classifier entities, such as tag

Also:

- unless explicitly specified, the visibility of an image should be the same as

its parent folder * the visibility of a comment should be the same as the commented entity * If there is no parent entity, the default visibility is **authenticated**.

Regarding write security, that’s much easier:

- anonymous can’t write anything
- authenticated users can only add comment
- managers will add the remaining stuff

Now, let’s implement that!

Proper security in CubicWeb is done **at the schema level**, so you don’t have to bother with it in views: users will only see what they can see automatically.

Step 1: configuring security into the schema

In the schema, you can grant access according to:

- groups
- to some RQL expressions: users get access if the expression returns some results

To implement the read security defined earlier, groups are not enough, we'll need some RQL expression. Here is the idea:

- add a *visibility* attribute on *Folder*, *File* and *Comment*, which may be one of the value explained above
- add a *may_be_read_by* relation from *Folder*, *File* and *Comment* to *users*, which will define who can see the entity
- security propagation will be done in hooks

Note: What makes *visibility* an attribute and not a relation is that its object is a primitive type, here *String*.

Other builtin primitives are *String*, *Int*, *BigInt*, *Float*, *Decimal*, *Boolean*, *Date*, *Datetime*, *Time*, *Interval*, *Byte* and *Password* and for more information read *Entity type*

So the first thing to do is to modify my cube's `schema.py` to define those relations:

```
from yams.constraints import StaticVocabularyConstraint

class visibility(RelationDefinition):
    subject = ('Folder', 'File', 'Comment')
    object = 'String'
    constraints = [StaticVocabularyConstraint(('public', 'authenticated',
                                              'restricted', 'parent'))]

    default = 'parent'
    cardinality = '11' # required

class may_be_read_by(RelationDefinition):
    __permissions__ = {
        'read': ('managers', 'users'),
        'add': ('managers',),
        'delete': ('managers',),
    }

    subject = ('Folder', 'File', 'Comment',)
    object = 'CWUser'
```

We can note the following points:

- we've added a new *visibility* attribute to *Folder*, *File*, *Image* and *Comment* using a *RelationDefinition*
- *cardinality = '11'* means this attribute is required. This is usually hidden under the *required* argument given to the *String* constructor, but we can rely on this here (same thing for *StaticVocabularyConstraint*, which is usually hidden by the *vocabulary* argument)
- the *parent* possible value will be used for visibility propagation
- think to secure the *may_be_read_by* permissions, else any user can add/delete it by default, which somewhat breaks our security model...

Now, we should be able to define security rules in the schema, based on these new attribute and relation. Here is the code to add to `schema.py`:

```
from cubicweb.schema import ERQLEExpression

VISIBILITY_PERMISSIONS = {
    'read': ( 'managers',
              ERQLEExpression('X visibility "public"'),
              ERQLEExpression('X may_be_read_by U')),
    'add': ( 'managers', ),
    'update': ( 'managers', 'owners', ),
    'delete': ( 'managers', 'owners', ),
}

AUTH_ONLY_PERMISSIONS = {
    'read': ( 'managers', 'users', ),
    'add': ( 'managers', ),
    'update': ( 'managers', 'owners', ),
    'delete': ( 'managers', 'owners', ),
}

CLASSIFIERS_PERMISSIONS = {
    'read': ( 'managers', 'users', 'guests', ),
    'add': ( 'managers', ),
    'update': ( 'managers', 'owners', ),
    'delete': ( 'managers', 'owners', ),
}

from cubicweb_folder.schema import Folder
from cubicweb_file.schema import File
from cubicweb_comment.schema import Comment
from cubicweb_person.schema import Person
from cubicweb_tag.schema import Tag

Folder.__permissions__ = VISIBILITY_PERMISSIONS
File.__permissions__ = VISIBILITY_PERMISSIONS
Comment.__permissions__ = VISIBILITY_PERMISSIONS.copy()
Comment.__permissions__['add'] = ( 'managers', 'users', )
Person.__permissions__ = AUTH_ONLY_PERMISSIONS
Tag.__permissions__ = CLASSIFIERS_PERMISSIONS
```

What's important in there:

- `VISIBILITY_PERMISSIONS` provides read access to managers group, if `visibility` attribute's value is 'public', or if user (designed by the 'U' variable in the expression) is linked to the entity (the 'X' variable) through the `may_be_read_by` permission
- we modify permissions of the entity types we use by importing them and modifying their `__permissions__` attribute
- notice the `.copy()`: we only want to modify 'add' permission for `Comment`, not for all entity types using `VISIBILITY_PERMISSIONS`!
- the remaining part of the security model is done using regular groups:
 - `users` is the group to which all authenticated users will belong
 - `guests` is the group of anonymous users

Step 2: security propagation in hooks

To fulfill the requirements defined earlier, we have to implement:

Also, unless explicitly specified, visibility of an image should be the same as its parent folder, as well as visibility of a comment should be the same as the commented entity.

This kind of *active* rule will be done using CubicWeb's hook system. Hooks are triggered on database events such as addition of a new entity or relation.

The tricky part of the requirement is in *unless explicitly specified*, notably because when the entity is added, we don't know yet its 'parent' entity (e.g. Folder of an File, File commented by a Comment). To handle such things, CubicWeb provides *Operation*, which allow to schedule things to do at commit time.

In our case we will:

- on entity creation, schedule an operation that will set default visibility
- when a *parent* relation is added, propagate parent's visibility unless the child already has a visibility set

Here is the code in cube's `hooks.py`:

```
from cubicweb.predicates import is_instance
from cubicweb.server import hook

class SetVisibilityOp(hook.DataOperationMixin, hook.Operation):

    def precommit_event(self):
        for eid in self.get_data():
            entity = self.cnx.entity_from_eid(eid)

            if entity.visibility == 'parent':
                entity.cw_set(visibility=u'authenticated')

class SetVisibilityHook(hook.Hook):
    __regid__ = 'sytweb.setvisibility'
    __select__ = hook.Hook.__select__ & is_instance('Folder', 'File', 'Comment')
    events = ('after_add_entity',)

    def __call__(self):
        SetVisibilityOp.get_instance(self._cw).add_data(self.entity.eid)

class SetParentVisibilityHook(hook.Hook):
    __regid__ = 'sytweb.setparentvisibility'
    __select__ = hook.Hook.__select__ & hook.match_rtype('filed_under', 'comments')
    events = ('after_add_relation',)

    def __call__(self):
        parent = self._cw.entity_from_eid(self.eidto)
        child = self._cw.entity_from_eid(self.eidfrom)

        if child.visibility == 'parent':
            child.cw_set(visibility=parent.visibility)
```

Notice:

- hooks are application objects, hence have selectors that should match entity or relation types to which the hook applies. To match a relation type, we use the hook specific *match_rtype* selector.
- usage of *DataOperationMixin*: instead of adding an operation for each added entity, *DataOperationMixin* allows to create a single one and to store entity's eids to be processed in the transaction data. This is a good practice to avoid heavy operations manipulation cost when creating a lot of entities in the same transaction.
- the *precommit_event* method of the operation will be called at transaction's commit time.
- in a hook, *self.cw* is the repository session, not a web request as usually in views
- according to hook's event, you have access to different attributes on the hook instance. Here:
 - *self.entity* is the newly added entity on 'after_add_entity' events
 - *self.eidfrom* / *self.eidto* are the eid of the subject / object entity on 'after_add_relation' events (you may also get the relation type using *self.rtype*)

The *parent* visibility value is used to tell “propagate using parent security” because we want that attribute to be required, so we can't use *None* value else we'll get an error before we get any chance to propagate...

Now, we also want to propagate the *may_be_read_by* relation. Fortunately, CubicWeb provides some base hook classes for such things, so we only have to add the following code to *hooks.py*:

```
# relations where the "parent" entity is the subject
S_RELS = set()
# relations where the "parent" entity is the object
O_RELS = set(('filed_under', 'comments',))

class AddEntitySecurityPropagationHook(hook.PropagateRelationHook):
    """propagate permissions when new entity are added"""
    __regid__ = 'sytweb.addentity_security_propagation'
    __select__ = (hook.PropagateRelationHook.__select__
                  & hook.match_rtype_sets(S_RELS, O_RELS))
    main_rtype = 'may_be_read_by'
    subject_relations = S_RELS
    object_relations = O_RELS

class AddPermissionSecurityPropagationHook(hook.PropagateRelationAddHook):
    """propagate permissions when new entity are added"""
    __regid__ = 'sytweb.addperm_security_propagation'
    __select__ = (hook.PropagateRelationAddHook.__select__
                  & hook.match_rtype('may_be_read_by',))
    subject_relations = S_RELS
    object_relations = O_RELS

class DelPermissionSecurityPropagationHook(hook.PropagateRelationDelHook):
    __regid__ = 'sytweb.delperm_security_propagation'
    __select__ = (hook.PropagateRelationDelHook.__select__
                  & hook.match_rtype('may_be_read_by',))
    subject_relations = S_RELS
    object_relations = O_RELS
```

- the *AddEntitySecurityPropagationHook* will propagate the relation when *filed_under* or *comments* relations are added
 - the *S_RELS* and *O_RELS* set as well as the *match_rtype_sets* selector are used here so that if my cube is used by another one, it'll be able to configure security propagation by simply adding relation to one of the two sets.
- the two others will propagate permissions changes on parent entities to children entities

Step 3: testing our security

Security is tricky. Writing some tests for it is a very good idea. You should even write them first, as Test Driven Development recommends!

Here is a small test case that will check the basis of our security model, in `test/test_sytweb.py`:

```
from cubicweb.devtools import testlib
from cubicweb import Binary

class SecurityTC(testlib.CubicWebTC):

    def test_visibility_propagation(self):
        with self.admin_access.repo_cnx() as cnx:
            # create a user for later security checks
            toto = self.create_user(cnx, 'toto')

            cnx.commit()

            # init some data using the default manager connection
            folder = cnx.create_entity('Folder',
                                      name=u'restricted',
                                      visibility=u'restricted')
            photo1 = cnx.create_entity('File',
                                      data_name=u'photo1.jpg',
                                      data=Binary(b'xxx'),
                                      filed_under=folder)

            cnx.commit()

            # visibility propagation
            self.assertEqual(photo1.visibility, 'restricted')

            # unless explicitly specified
            photo2 = cnx.create_entity('File',
                                      data_name=u'photo2.jpg',
                                      data=Binary(b'xxx'),
                                      visibility=u'public',
                                      filed_under=folder)

            cnx.commit()

            self.assertEqual(photo2.visibility, 'public')
```

(continues on next page)

(continued from previous page)

```

with self.new_access('toto').repo_cnx() as cnx:
    # test security
    self.assertEqual(1, len(cnx.execute('File X'))) # only the public one
    self.assertEqual(0, len(cnx.execute('Folder X'))) # restricted...

with self.admin_access.repo_cnx() as cnx:
    # may_be_read_by propagation
    folder = cnx.entity_from_eid(folder.eid)
    folder.cw_set(may_be_read_by=toto)

    cnx.commit()

with self.new_access('toto').repo_cnx() as cnx:
    photo1 = cnx.entity_from_eid(photo1.eid)

    self.failUnless(photo1.may_be_read_by)

    # test security with permissions
    self.assertEqual(2, len(cnx.execute('File X'))) # now toto has access to
↪photo2
    self.assertEqual(1, len(cnx.execute('Folder X'))) # and to restricted folder

if __name__ == '__main__':
    from unittest import main
    main()

```

It's not complete, but shows most things you'll want to do in tests: adding some content, creating users and connecting as them in the test, etc...

To run it type:

```

$ python3 test/test_sytweb.py
=====
-> creating tables [=====]
-> inserting default user and default groups.
-> storing the schema in the database [=====]
-> database for instance data initialized.
.
-----
Ran 1 test in 22.547s

OK

```

The first execution is taking time, since it creates a sqlite database for the test instance. The second one will be much quicker:

```

$ python3 test/test_sytweb.py
=====
.
-----
Ran 1 test in 2.662s

```

(continues on next page)

(continued from previous page)

OK

If you do some changes in your schema, you'll have to force regeneration of that database. You do that by removing the tmpdb files before running the test:

```
$ rm data/database/tmpdb*
```

Step 4: writing the migration script and migrating the instance

Prior to those changes, I created an instance, fed it with some data, so I don't want to create a new one, but to migrate the existing one. Let's see how to do that.

Migration commands should be put in the cube's `migration` directory, in a file named `<X.Y.Z>_Any.py` ('Any' being there mostly for historical reasons and '`<X.Y.Z>`' being the version number of the cube we are going to release.)

Here I'll create a `migration/0.2.0_Any.py` file containing the following instructions:

```
add_relation_type('may_be_read_by')
add_relation_type('visibility')
sync_schema_props_perms()
```

Then I update the version number in the cube's `__pkginfo__.py` to 0.2.0. And that's it! Those instructions will:

- update the instance's schema by adding our two new relations and update the underlying database tables accordingly (the first two instructions)
- update schema's permissions definition (the last instruction)

To migrate my instance I simply type:

```
cubicweb-ctl upgrade sytweb_instance
```

You'll then be asked some questions to do the migration step by step. You should say YES when it asks if a backup of your database should be done, so you can get back to initial state if anything goes wrong...

Storing images on the file-system

Step 1: configuring the BytesFileSystem storage

To avoid cluttering my database, and to ease file manipulation, I don't want them to be stored in the database. I want to be able create File entities for some files on the server file system, where those file will be accessed to get entities data. To do so I've to set a custom BytesFileSystemStorage storage for the File 'data' attribute, which hold the actual file's content.

Since the function to register a custom storage needs to have a repository instance as first argument, we've to call it in a server startup hook. So I added in `cubicweb_sytweb/hooks.py` :

```
from os import makedirs
from os.path import join, exists

from cubicweb.server import hook
from cubicweb.server.sources import storages
```

(continues on next page)

(continued from previous page)

```
class ServerStartupHook(hook.Hook):
    __regid__ = 'sytweb.serverstartup'
    events = ('server_startup', 'server_maintenance')

    def __call__(self):
        bfssdir = join(self.repo.config.appdatahome, 'bfss')
        if not exists(bfssdir):
            mkdirs(bfssdir)
            print('created', bfssdir)
        storage = storages.BytesFileSystemStorage(bfssdir)
        storages.set_attribute_storage(self.repo, 'File', 'data', storage)
```

Note:

- how we built the hook's registry identifier (`__regid__`): you can introduce 'namespaces' by using there python module like naming identifiers. This is especially important for hooks where you usually want a new custom hook, not overriding / specializing an existant one, but the concept may be applied to any application objects
 - we catch two events here: "server_startup" and "server_maintenance". The first is called on regular repository startup (eg, as a server), the other for maintenance task such as shell or upgrade. In both cases, we need to have the storage set, else we'll be in trouble...
 - the path given to the storage is the place where file added through the ui (or in the database before migration) will be located
 - beware that by doing this, you can't anymore write queries that will try to restrict on File *data* attribute. Hopefully we don't do that usually on file's content or more generally on attributes for the Bytes type
-

Now, if you've already added some photos through the web ui, you'll have to migrate existing data so file's content will be stored on the file-system instead of the database. There is a migration command to do so, let's run it in the cubicweb shell (in real life, you would have to put it in a migration script as we have seen last time):

```
$ cubicweb-ctl shell sytweb_instance
entering the migration python shell
just type migration commands or arbitrary python code and type ENTER to execute it
type "exit" or Ctrl-D to quit the shell and resume operation
>>> storage_changed('File', 'data')
[=====]
```

That's it. Now, files added through the web ui will have their content stored on the file-system, and you'll also be able to import files from the file-system as explained in the next part.

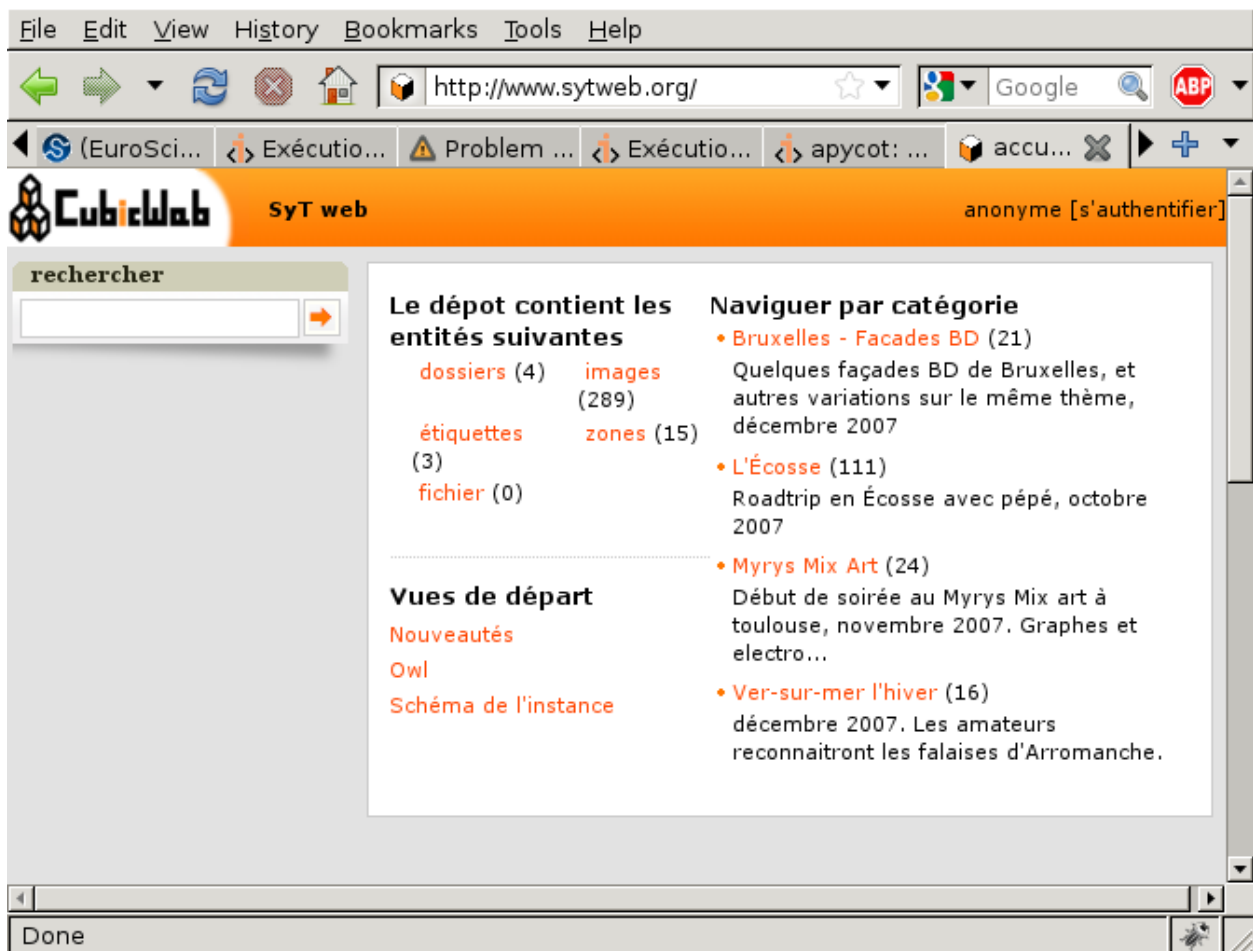
Step 2: importing some data into the instance

Hey, we start to have some nice features, let us give a try to this new web site. For instance if I have a 'photos/201005WePyrenees' containing pictures for a particular event, I can import it to my web site by typing

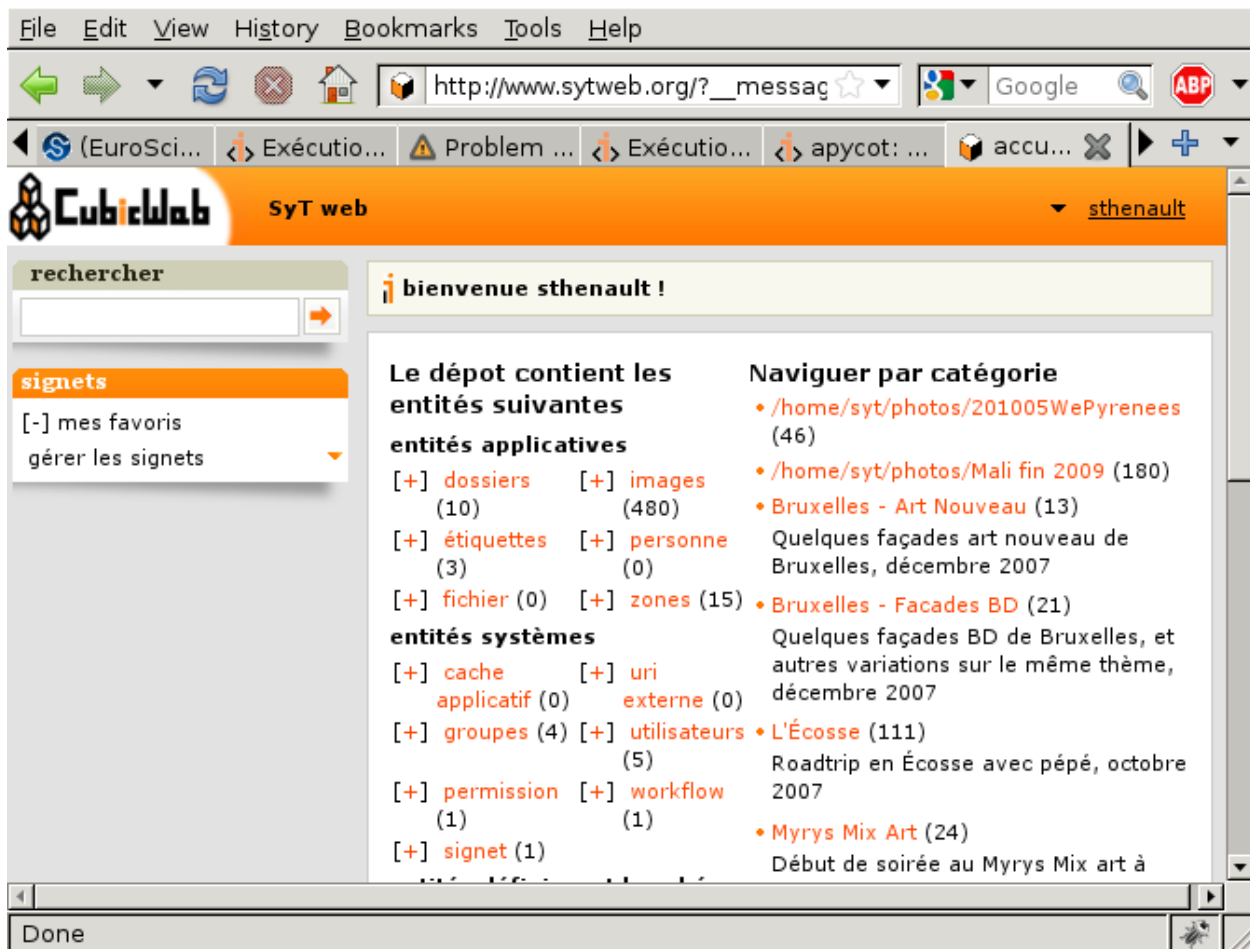
```
$ cubicweb-ctl fsimport -F sytweb_instance photos/201005WePyrenees/
** importing directory /home/syt/photos/201005WePyrenees
importing IMG_8314.JPG
importing IMG_8274.JPG
importing IMG_8286.JPG
importing IMG_8308.JPG
importing IMG_8304.JPG
```

Note: The -F option means that folders should be mapped, hence my photos will be linked to a Folder entity corresponding to the file-system folder.

Let's take a look at the web ui:

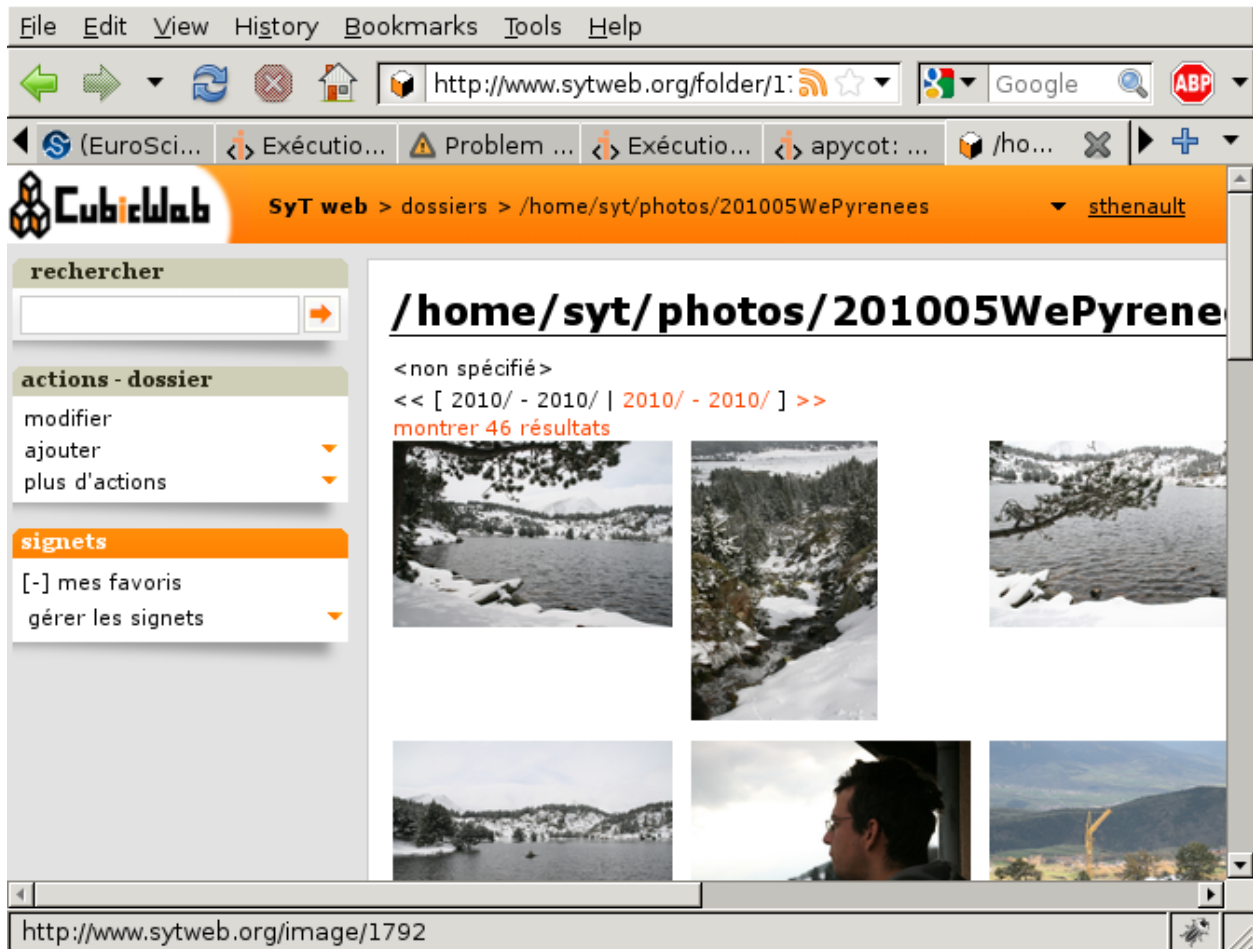


Nothing different, I can't see the new folder... But remember our security model! By default, files are only accessible to authenticated users, and I'm looking at the site as anonymous, e.g. not authenticated. If I login, I can now see:



Yeah, it's there! You will notice that I can see some entities as well as folders and images the anonymous user can't. It just works **everywhere in the ui** since it's handled at the repository level, thanks to our security model.

Now if I click on the recently inserted folder, I can see



Great! There is even my pictures in the folder. I can now give to this folder a nicer name (provided I don't intend to import from it anymore, else already imported photos will be reimported), change permissions, title for some pictures, etc... Having a good content is much more difficult than having a good web site ;)

Conclusion

We started to see here an advanced feature of our repository: the ability to store some parts of our data-model into a custom storage, outside the database. There is currently only the `BytesFileSystemStorage` available, but you can expect to see more coming in a near future (or write your own!).

Also, we can now start to feed our web-site with some nice pictures! The site isn't perfect (far from it actually) but it's usable, and we can now start using it and improve it on the way. The Incremental Cubic Way :)

Let's make it more user friendly

Step 1: let's improve site's usability for our visitors

The first thing I've noticed is that people to whom I send links to photos with some login/password authentication get lost, because they don't grasp they have to login by clicking on the 'authenticate' link. That's much probably because they only get a 404 when trying to access an unauthorized folder, and the site doesn't make clear that 1. you're not authenticated, 2. you could get more content by authenticating yourself.

So, to improve this situation, I decided that I should:

- make a login box appears for anonymous, so they see at a first glance a place to put the login / password information I provided
- customize the 404 page, proposing to login to anonymous.

Here is the code, samples from my cube's `views.py` file:

```
from cubicweb import _
from cubicweb.web import component
from cubicweb.web.views import error
from cubicweb.predicates import anonymous_user

class FourOhFour(error.FourOhFour):
    __select__ = error.FourOhFour.__select__ & anonymous_user()

    def call(self):
        self.w(u"<h1>%s</h1>" % self._cw._('this resource does not exist'))
        self.w(u"<p>%s</p>" % self._cw._('have you tried to login?'))

class LoginBox(component.CtxComponent):
    """display a box containing links to all startup views"""
    __regid__ = 'sytweb.loginbox'
    __select__ = component.CtxComponent.__select__ & anonymous_user()

    title = _('Authenticate yourself')
    order = 70

    def render_body(self, w):
        cw = self._cw
        form = cw.vreg['forms'].select('logform', cw)
        form.render(w=w, table_class='', display_progress_div=False)
```

The first class provides a new specific implementation of the default page you get on 404 error, to display an adapted message to anonymous user.

Note: Thanks to the selection mechanism, it will be selected for anonymous user, since the additional `anonymous_user()` selector gives it a higher score than the default, and not for authenticated since this selector will return 0 in such case (hence the object won't be selectable)

The second class defines a simple box, that will be displayed by default with boxes in the left column, thanks to default `component.CtxComponent` selector. The HTML is written to match default CubicWeb boxes style. The code fetch the actual login form and render it.



Fig. 1: The login box and the custom 404 page for an anonymous visitor (translated in french)

Step 2: providing a custom index page

Another thing we can easily do to improve the site is... A nicer index page (e.g. the first page you get when accessing the web site)! The default one is quite intimidating (that should change in a near future). I will provide a much simpler index page that simply list available folders (e.g. photo albums in that site).

Here is the code, samples from my cube's `views.py` file:

```
from cubicweb.web.views import startup

class IndexView(startup.IndexView):
    def call(self, **kwargs):
        self.w(u'<div>\n')
        if self._cw.cnx.session.anonymous_session:
            self.w(u'<h4>%s</h4>\n' % self._cw._('Public Albums'))
        else:
            self.w(u'<h4>%s</h4>\n' % self._cw._('Albums for %s') % self._cw.user.login)
        self._cw.vreg['views'].select('tree', self._cw).render(w=self.w)
        self.w(u'</div>\n')

def registration_callback(vreg):
    vreg.register_all(globals().values(), __name__, (IndexView,))
    vreg.register_and_replace(IndexView, startup.IndexView)
```

As you can see, we override the default index view found in `cubicweb.web.views.startup`, getting back nothing but its identifier and selector since we override the top level view's `call` method.

Note: in that case, we want our index view to **replace** the existing one. To do so we've to implements the `registration_callback` function, in which we tell to register everything in the module *but* our `IndexView`, then we register it instead of the former index view.

Also, we added a title that tries to make it more evident that the visitor is authenticated, or not. Hopefully people will get it now!

SyT web

Naviguer par type d'entité

dossiers (4) fichiers (302)
étiquettes (12) zones (25)

Vues de départ

Nouveautés
Owl
Schéma de l'instance

Naviguer par catégorie

Bruxelles - Facades BD (21)
Quelques façades BD de Bruxelles, et autres variations sur le même thème, décembre 2007

L'Écosse (111)
Roadtrip en Écosse avec pépé, octobre 2007

Myrys Mix Art (24)
Début de soirée au Myrys Mix art à toulouse, novembre 2007. Graphes et electro...

Ver-sur-mer l'hiver (16)
décembre 2007. Les amateurs reconnaîtront les falaises d'Arromanche.

Fig. 2: The default index page

Albums publics

Bruxelles - Facades BD (21)
Quelques façades BD de Bruxelles, et autres variations sur le même thème, décembre 2007

L'Écosse (111)
Roadtrip en Écosse avec pépé, octobre 2007

Myrys Mix Art (24)
Début de soirée au Myrys Mix art à toulouse, novembre 2007. Graphes et electro...

Ver-sur-mer l'hiver (16)
décembre 2007. Les amateurs reconnaîtront les falaises d'Arromanche.

Fig. 3: Our simpler, less intimidating, index page (still translated in french)

Step 3: more navigation improvements

There are still a few problems I want to solve...

- Images in a folder are displayed in a somewhat random order. I would like to have them ordered by file's name (which will usually, inside a given folder, also result ordering photo by their date and time)
- When clicking a photo from an album view, you've to get back to the gallery view to go to the next photo. This is pretty annoying...
- Also, when viewing an image, there is no clue about the folder to which this image belongs to.

I will first try to explain the ordering problem. By default, when accessing related entities by using the ORM's API, you should get them ordered according to the target's class `cw_fetch_order`. If we take a look at the file cube's schema, we can see:

```
class File(AnyEntity):
    """customized class for File entities"""
    __regid__ = 'File'
    fetch_attrs, cw_fetch_order = fetch_config(['data_name', 'title'])
```

By default, `fetch_config` will return a `cw_fetch_order` method that will order on the first attribute in the list. So, we could expect to get files ordered by their name. But we don't. What's up doc?

The problem is that files are related to folder using the `filed_under` relation. And that relation is ambiguous, eg it can lead to *File* entities, but also to *Folder* entities. In such case, since both entity types doesn't share the attribute on which we want to sort, we'll get linked entities sorted on a common attribute (usually `modification_date`).

To fix this, we've to help the ORM. We'll do this in the method from the *ITree* folder's adapter, used in the folder's primary view to display the folder's content. Here's the code, that I've put in our cube's `entities.py` file, since it's more logical stuff than view stuff:

```
from cubicweb_folder import entities as folder

class FolderITreeAdapter(folder.FolderITreeAdapter):

    def different_type_children(self, entities=True):
        rql = self.entity.cw_related_rql(self.tree_relation,
                                         self.parent_role, ('File',))
        rset = self._cw.execute(rql, {'x': self.entity.eid})

        if entities:
            return list(rset.entities())

        return rset

def registration_callback(vreg):
    vreg.register_and_replace(FolderITreeAdapter, folder.FolderITreeAdapter)
```

As you can see, we simply inherit from the adapter defined in the *folder* cube, then we override the `different_type_children` method to give a clue to the ORM's `cw_related_rql` method, that is responsible to generate the rql to get entities related to the folder by the `filed_under` relation (the value of the `tree_relation` attribute). The clue is that we only want to consider the *File* target entity type. By doing this, we remove the ambiguity and get back a RQL query that correctly order files by their `data_name` attribute.

Note:

- As seen earlier, we want to **replace** the folder's *ITree* adapter by our implementation, hence the custom *registration_callback* method.
-

Ouf. That one was tricky...

Now the easier parts. Let's start by adding some links on the file's primary view to see the previous / next image in the same folder. CubicWeb's provide a component that do exactly that. To make it appears, one have to be adaptable to the *IPrevNext* interface. Here is the related code sample, extracted from our cube's `views.py` file:

```
from cubicweb.predicates import is_instance
from cubicweb.web.views import navigation

class FileIPrevNextAdapter(navigation.IPrevNextAdapter):
    __select__ = is_instance('File')

    def previous_entity(self):
        rset = self._cw.execute('File F ORDERBY FDN DESC LIMIT 1 WHERE '
                                'X filed_under FOLDER, F filed_under FOLDER, '
                                'F data_name FDN, X data_name > FDN, X eid %(x)s',
                                {'x': self.entity.eid})

        if rset:
            return rset.get_entity(0, 0)

    def next_entity(self):
        rset = self._cw.execute('File F ORDERBY FDN ASC LIMIT 1 WHERE '
                                'X filed_under FOLDER, F filed_under FOLDER, '
                                'F data_name FDN, X data_name < FDN, X eid %(x)s',
                                {'x': self.entity.eid})

        if rset:
            return rset.get_entity(0, 0)
```

The *IPrevNext* interface implemented by the adapter simply consist in the *previous_entity* / *next_entity* methods, that should respectively return the previous / next entity or *None*. We make an RQL query to get files in the same folder, ordered similarly (eg by their *data_name* attribute). We set ascendant/descendant ordering and a strict comparison with current file's name (the "X" variable representing the current file).

Notice that this query supposes we wont have two files of the same name in the same folder, else things may go wrong. Fixing this is out of the scope of this tutorial. And as I would like to have at some point a smarter, context sensitive previous/next entity, I'll probably never fix this query (though if I had to, I would probably choosing to add a constraint in the schema so that we can't add two files of the same name in a folder).

One more thing: by default, the component will be displayed below the content zone (the one with the white background). You can change this in the site's properties through the ui, but you can also change the default value in the code by modifying the *context* attribute of the component *FileIPrevNextAdapter*:

```
navigation.NextPrevNavigationComponent.context = 'navcontentbottom'
```

Note: *context* may be one of 'navtop', 'navbottom', 'navcontenttop' or 'navcontentbottom'; the first two being outside the main content zone, the two others inside it.

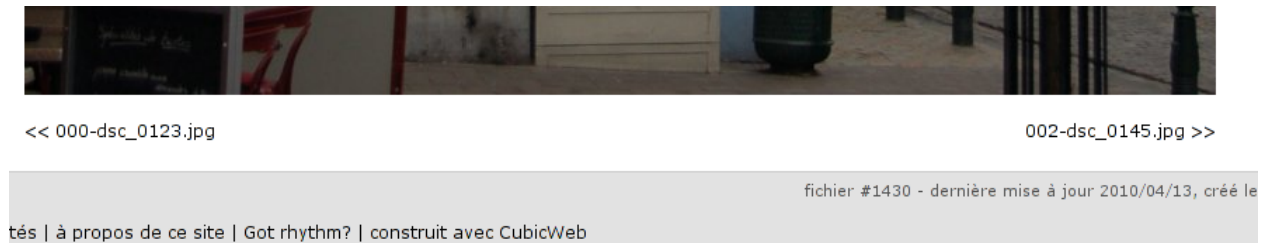


Fig. 4: The previous/next entity component, at the bottom of the main content zone.

Now, the only remaining stuff in my todo list is to see the file's folder. I'll use the standard breadcrumb component to do so. Similarly as what we've seen before, this component is controlled by the `IBreadCrumbs` interface, so we'll have to provide a custom adapter for *File* entity, telling the a file's parent entity is its folder:

```
from cubicweb.web.views import ibreadcrumbs

class FileIBreadCrumbsAdapter(ibreadcrumbs.IBreadCrumbsAdapter):
    __select__ = is_instance('File')

    def parent_entity(self):
        if self.entity.filed_under:
            return self.entity.filed_under[0]
```

In that case, we simply use attribute notation provided by the ORM to get the folder in which the current file (e.g. *self.entity*) is located.

Note: The `IBreadCrumbs` interface is a *breadcrumbs* method, but the default `IBreadCrumbsAdapter` provides a default implementation for it that will look at the value returned by its *parent_entity* method. It also provides a default implementation for this method for entities adapting to the *ITree* interface, but as our *File* doesn't, we've to provide a custom adapter.



Fig. 5: The breadcrumb component when on a file entity, now displaying parent folder.

Step 4: preparing the release and migrating the instance

Now that greatly enhanced our cube, it's time to release it to upgrade production site. I'll probably detail that process later, but I currently simply transfer the new code to the server running the web site.

However, I've still today some step to respect to get things done properly...

First, as I've added some translatable string, I've to run:

```
$ cubicweb-ctl i18ncube sytweb
```

To update the cube's gettext catalogs (the '.po' files under the cube's *i18n* directory). Once the above command is executed, I'll then update translations.

To see if everything is ok on my test instance, I do:

```
$ cubicweb-ctl i18ninstance sytweb_instance
$ cubicweb-ctl pyramid -D sytweb_instance
```

The first command compile i18n catalogs (e.g. generates '.mo' files) for my test instance. The second command start it in debug mode, so I can open my browser and navigate through the web site to see if everything is ok...

Note: In the 'cubicweb-ctl i18ncube' command, *sytweb* refers to the **cube**, while in the two other, it refers to the **instance** (if you can't see the difference, reread [CubicWeb's concept chapter](#)!).

Once I've checked it's ok, I simply have to bump the version number in the `__pkginfo__` module to trigger a migration once I'll have updated the code on the production site. I can check then check the migration is also going fine, by first restoring a dump from the production site, then upgrading my test instance.

To generate a dump from the production site:

```
$ cubicweb-ctl db-dump sytweb_instance
# if it's postgresql
pg_dump -Fc --username=syt --no-owner --file /home/syt/etc/cubicweb.d/sytweb/backup/
↳ tmpYIN0YI/system sytweb
# if it's sqlite
gzip -c /home/psycojoker/etc/cubicweb.d/sytweb_instance/sytweb_instance.sqlite
-> backup file /home/syt/etc/cubicweb.d/sytweb/backup/sytweb-2010-07-13_10-22-40.tar.gz
```

I can now get back the dump file (*sytweb-2010-07-13_10-22-40.tar.gz*) to my test machine (using *scp* for instance) to restore it and start migration:

```
$ cubicweb-ctl db-restore sytweb_instance /path/path/to/sytweb-2010-07-13_10-22-40.tar.gz
$ cubicweb-ctl upgrade sytweb_instance
```

You might have to answer some questions, as we've seen in [a previous part](#).

Now that everything is tested, I can transfer the new code to the production server, *pip install* CubicWeb and its dependencies, and eventually upgrade the production instance.

Building my photos web site with *CubicWeb* part V: let's make it even more user friendly

Step 1: tired of the default look?

OK... Now our site has its most desired features. But... I would like to make it look somewhat like *my* website. It is not www.cubicweb.org after all. Let's tackle this first!

The first thing we can do is to change the logo. There are various ways to achieve this. The easiest way is to put a `logo.png` file into the cube's `data` directory. As data files are looked at according to cubes order (CubicWeb resources coming last), that file will be selected instead of CubicWeb's one.

Note: As the location for static resources are cached, you'll have to restart your instance for this to be taken into account.

Though there are some cases where you don't want to use a `logo.png` file. For instance if it's a JPEG file. You can still change the logo by defining in the cube's `uiprops.py` file:

```
LOGO = data('logo.jpg')
```

Note: If the file `uiprops.py` doesn't exist in your cube, simply create it.

The `uiprops` machinery is used to define some static file resources, such as the logo, default Javascript / CSS files, as well as CSS properties (we'll see that later).

Note: This file is imported specifically by *CubicWeb*, with a predefined name space, containing for instance the `data` function, telling the file is somewhere in a cube or CubicWeb's data directory.

One side effect of this is that it can't be imported as a regular python module.

The nice thing is that in debug mode, change to a `uiprops.py` file are detected and then automatically reloaded.

Now, as it's a photos web-site, I would like to have a photo of mine as background... After some trials I won't detail here, I've found a working recipe explained [here](#). All I've to do is to override some stuff of the default CubicWeb user interface to apply it as explained.

The first thing to do to get the `` tag as first element after the `<body>` tag. If you know a way to avoid this by simply specifying the image in the CSS, tell me! The easiest way to do so is to override the `HTMLPageHeader` view, since that's the one that is directly called once the `<body>` has been written. How did I find this? By looking in the `cubiweb.web.views.basetemplates` module, since I know that global page layouts sits there. I could also have grep the "body" tag in `cubicweb.web.views...` Finding this was the hardest part. Now all I need is to customize it to write that `img` tag, as below in `views.py`:

```
from cubicweb.web.views import basetemplates

class HTMLPageHeader(basetemplates.HTMLPageHeader):
    # override this since it's the easier way to have our bg image
    # as the first element following <body>
    def call(self, **kwargs):
        self.w(u''
              % self._cw.datadir_url)
        super(HTMLPageHeader, self).call(**kwargs)
```

(continues on next page)

(continued from previous page)

```
def registration_callback(vreg):
    vreg.register_all(globals().values(), __name__, (HTMLPageHeader))
    vreg.register_and_replace(HTMLPageHeader, basetemplates.HTMLPageHeader)
```

As you may have guessed, my background image is in a `background.jpg` file in the cube's data directory, but there are still some things to explain to newcomers here:

- The `call()` method is there the main access point of the view. It's called by the view's `render()` method. It is not the only access point for a view, but this will be detailed later.
- Calling `self.w` writes something to the output stream. Except for binary views (which do not generate text), it *must* be passed an Unicode string.
- The proper way to get a file in data directory is to use the `datadir_url` attribute of the incoming request (e.g. `self.cw`).

I won't explain again the `registration_callback()` stuff, you should understand it now! If not, go back to [previous post in the series](#) :)

Fine. Now all I've to do is to add a bit of CSS to get it to behave nicely (which is not the case at all for now). I'll put all this in a `cubes.sytweb.css` file, stored as usual in our data directory:

```
/* fixed full screen background image
 * as explained on http://webdesign.about.com/od/css3/f/blfaqbgsize.htm
 *
 * syt update: set z-index=0 on the img instead of z-index=1 on div#page & co to
 * avoid pb with the user actions menu
 */
img#bg-image {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    z-index: 0;
}

div#page, table#header, div#footer {
    background: transparent;
    position: relative;
}

/* add some space around the logo
 */
img#logo {
    padding: 5px 15px 0px 15px;
}

/* more dark font for metadata to have a chance to see them with the background
 * image
 */
div.metadata {
```

(continues on next page)

(continued from previous page)

```
color: black;
}
```

You can see here stuff explained in the cited page, with only a slight modification explained in the comments, plus some additional rules to make things somewhat cleaner:

- a bit of padding around the logo
- darker metadata which appears by default below the content (the white frame in the page)

To get this CSS file used everywhere in the site, I have to modify the `uiprops.py` file introduced above:

```
STYLESHEETS = sheet['STYLESHEETS'] + [data('cubes.sytweb.css')]
```

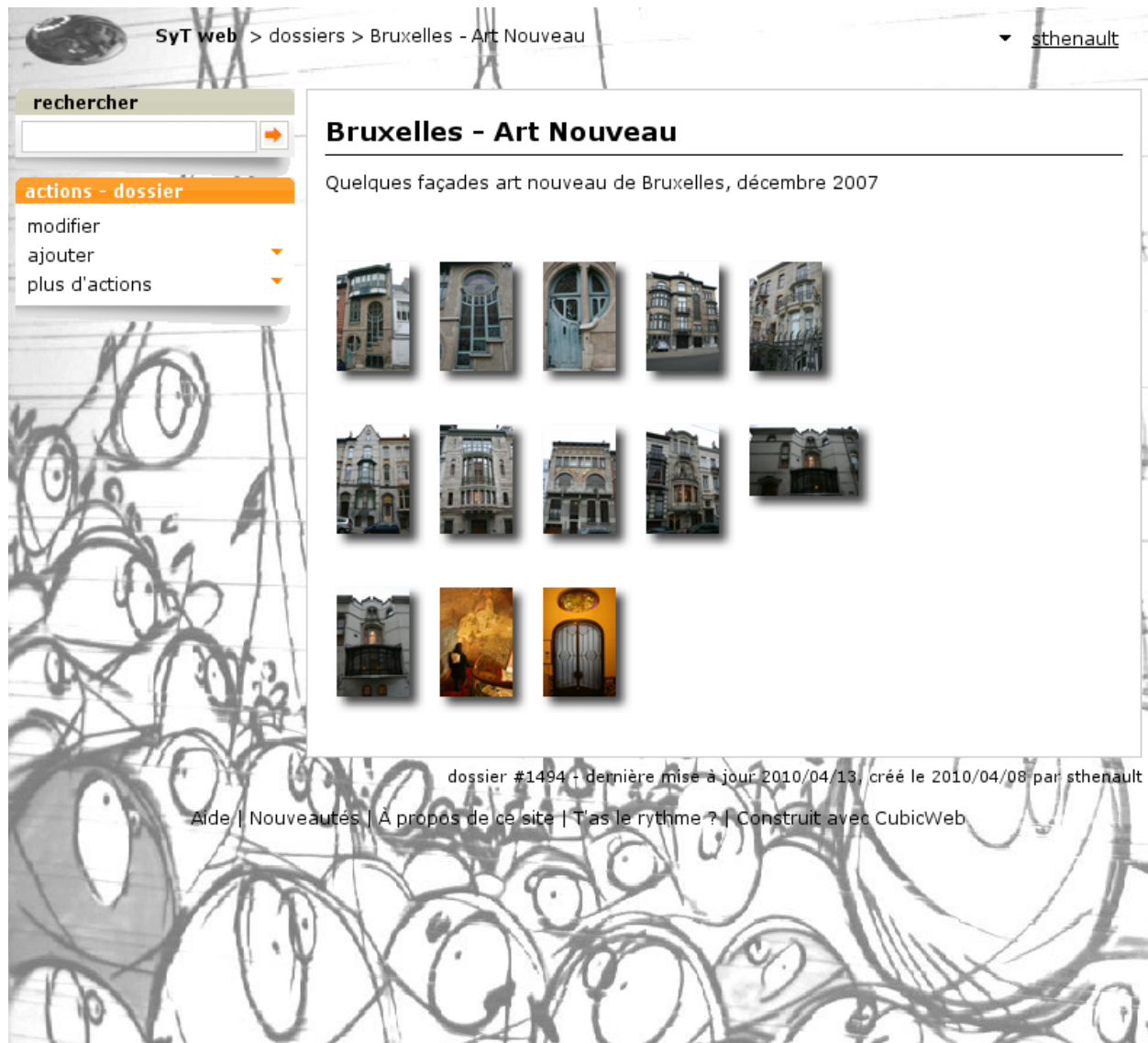
Note: *sheet* is another predefined variable containing values defined by already process `:file: `uiprops.py`` file, notably the CubicWeb's one.

Here we simply want our CSS in addition to CubicWeb's base CSS files, so we redefine the *STYLESHEETS* variable to existing CSS (accessed through the *sheet* variable) with our one added. I could also have done:

```
sheet['STYLESHEETS'].append(data('cubes.sytweb.css'))
```

But this is less interesting since we don't see the overriding mechanism...

At this point, the site should start looking good, the background image being resized to fit the screen.



The final touch: let's customize CubicWeb's CSS to get less orange... By simply adding

```
contextualBoxTitleBg = incontextBoxTitleBg = '#AAAAAA'
```

and reloading the page we've just seen, we now have a nice greyed box instead of the orange one:



This is because CubicWeb's CSS include some variables which are expanded by values defined in `uiprops.py` file. In our case we controlled the properties of the CSS `background` property of boxes with CSS class `contextualBoxTitleBg` and `incontextBoxTitleBg`.

Step 2: configuring boxes

Boxes present to the user some ways to use the application. Let's first do a few user interface tweaks in our `views.py` file:

```
from cubicweb.predicates import none_rset
from cubicweb.web.views import bookmark
from cubicweb_zone import views as zone
from cubicweb_tag import views as tag

# change bookmarks box selector so it's only displayed on startup views
bookmark.BookmarksBox.__select__ = bookmark.BookmarksBox.__select__ & none_rset()
# move zone box to the left instead of in the context frame and tweak its order
zone.ZoneBox.context = 'left'
zone.ZoneBox.order = 100
# move tags box to the left instead of in the context frame and tweak its order
tag.TagsBox.context = 'left'
tag.TagsBox.order = 102
# hide similarity box, not interested
tag.SimilarityBox.visible = False
```

The idea is to move all boxes in the left column, so we get more space for the photos. Now, serious things: I want a box similar to the tags box but to handle the *Person displayed_on File* relation. We can do this simply by adding a `AjaxEditRelationCtxComponent` subclass to our views, as below:

```
from cubicweb import _
from logilab.common.decorators import monkeypatch
from cubicweb import ValidationError
```

(continues on next page)

(continued from previous page)

```

from cubicweb.web.views import uicfg, component
from cubicweb.web.views import basecontrollers

# hide displayed_on relation using uicfg since it will be displayed by the box below
uicfg.primaryview_section.tag_object_of('*', 'displayed_on', '*'), 'hidden')

class PersonBox(component.AjaxEditRelationCtxComponent):
    __regid__ = 'syweb.displayed-on-box'
    # box position
    order = 101
    context = 'left'
    # define relation to be handled
    rtype = 'displayed_on'
    role = 'object'
    target_etype = 'Person'
    # messages
    added_msg = _('person has been added')
    removed_msg = _('person has been removed')
    # bind to js_* methods of the json controller
    fname_vocabulary = 'unrelated_persons'
    fname_validate = 'link_to_person'
    fname_remove = 'unlink_person'

@monkeypatch(basecontrollers.JsonController)
@basecontrollers.jsonize
def js_unrelated_persons(self, eid):
    """return tag unrelated to an entity"""
    rql = "Any F + ' ' + S WHERE P surname S, P firstname F, X eid %(x)s, NOT P_
    ↪displayed_on X"
    return [name for (name,) in self._cw.execute(rql, {'x' : eid})]

@monkeypatch(basecontrollers.JsonController)
def js_link_to_person(self, eid, people):
    req = self._cw
    for name in people:
        name = name.strip().title()
        if not name:
            continue
        try:
            firstname, surname = name.split(None, 1)
        except:
            raise ValidationError(eid, {'displayed_on', 'object': 'provide <first name>
    ↪ <surname>'})
        rset = req.execute('Person P WHERE '
                           'P firstname %(firstname)s, P surname %(surname)s',
                           locals())
        if rset:
            person = rset.get_entity(0, 0)

```

(continues on next page)

(continued from previous page)

```

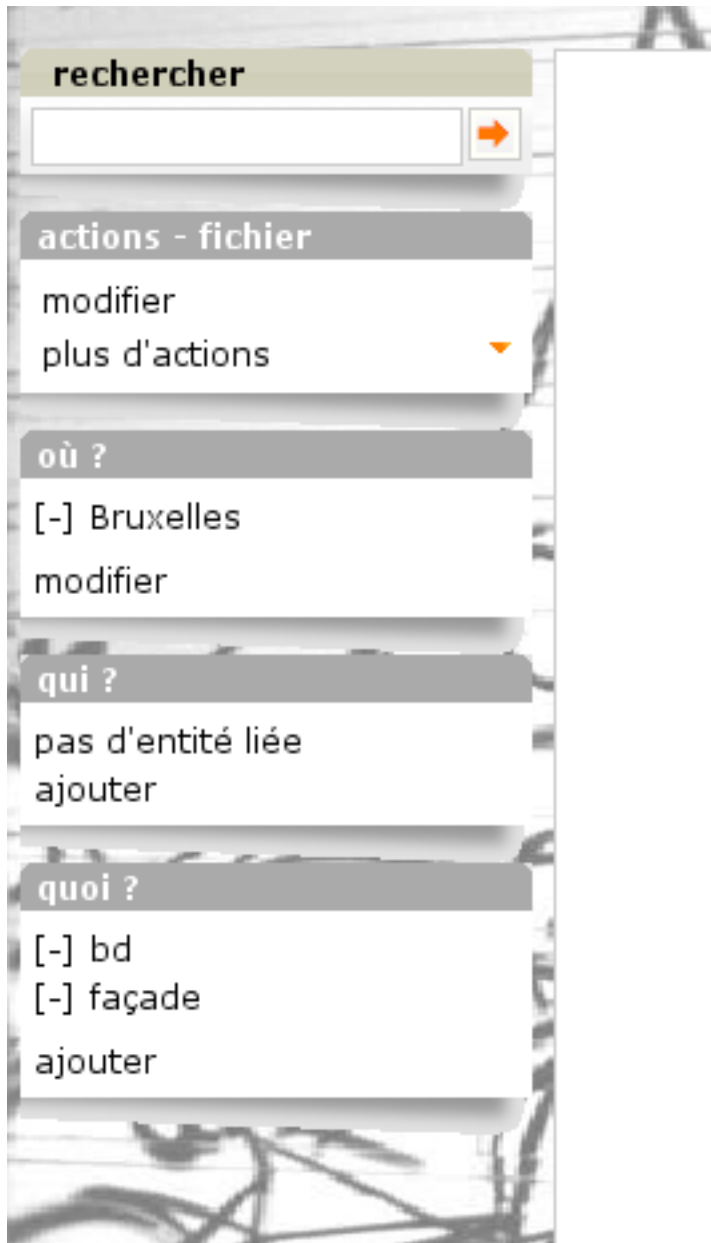
else:
    person = req.create_entity('Person', firstname=firstname,
                               surname=surname)
    req.execute('SET P displayed_on X WHERE '
               'P eid %(p)s, X eid %(x)s, NOT P displayed_on X',
               {'p': person.eid, 'x': eid})

@monkeypatch(basecontrollers.JsonController)
def js_unlink_person(self, eid, personeid):
    self._cw.execute('DELETE P displayed_on X WHERE P eid %(p)s, X eid %(x)s',
                     {'p': personeid, 'x': eid})

```

You basically subclass to configure with some class attributes. The *fname_** attributes give the name of methods that should be defined on the json control to make the AJAX part of the widget work: one to get the vocabulary, one to add a relation and another to delete a relation. These methods must start by a *js_* prefix and are added to the controller using the *@monkeypatch* decorator. In my case, the most complicated method is the one which adds a relation, since it tries to see if the person already exists, and else automatically create it, assuming the user entered “firstname surname”.

Let’s see how it looks like on a file primary view:



Great, it's now as easy for me to link my pictures to people than to tag them. Also, visitors get a consistent display of these two pieces of information.

Note: The ui component system has been refactored in [CubicWeb 3.10](#), which also introduced the `AjaxEditRelationCtxComponent` class.

Step 3: configuring facets

The last feature we'll add today is facet configuration. If you access to the '/file' url, you'll see a set of 'facets' appearing in the left column. Facets provide an intuitive way to build a query incrementally, by proposing to the user various way to restrict the result set. For instance CubicWeb proposes a facet to restrict based on who created an entity; the tag cube proposes a facet to restrict based on tags; the zoe cube a facet to restrict based on geographical location, and so on. In that gist, I want to propose a facet to restrict based on the people displayed on the picture. To do so, there are various classes in the `cubicweb.web.facet` module which simply have to be configured using class attributes as we've done for the box. In our case, we'll define a subclass of `RelationFacet`.

Note: Since that's ui stuff, we'll continue to add code below to our `views.py` file. Though we begin to have a lot of various code there, so it's may be a good time to split our views module into submodules of a `view` package. In our case of a simple application (glue) cube, we could start using for instance the layout below:

```
views/__init__.py    # uicfg configuration, facets
views/layout.py      # header/footer/background stuff
views/components.py  # boxes, adapters
views/pages.py       # index view, 404 view
```

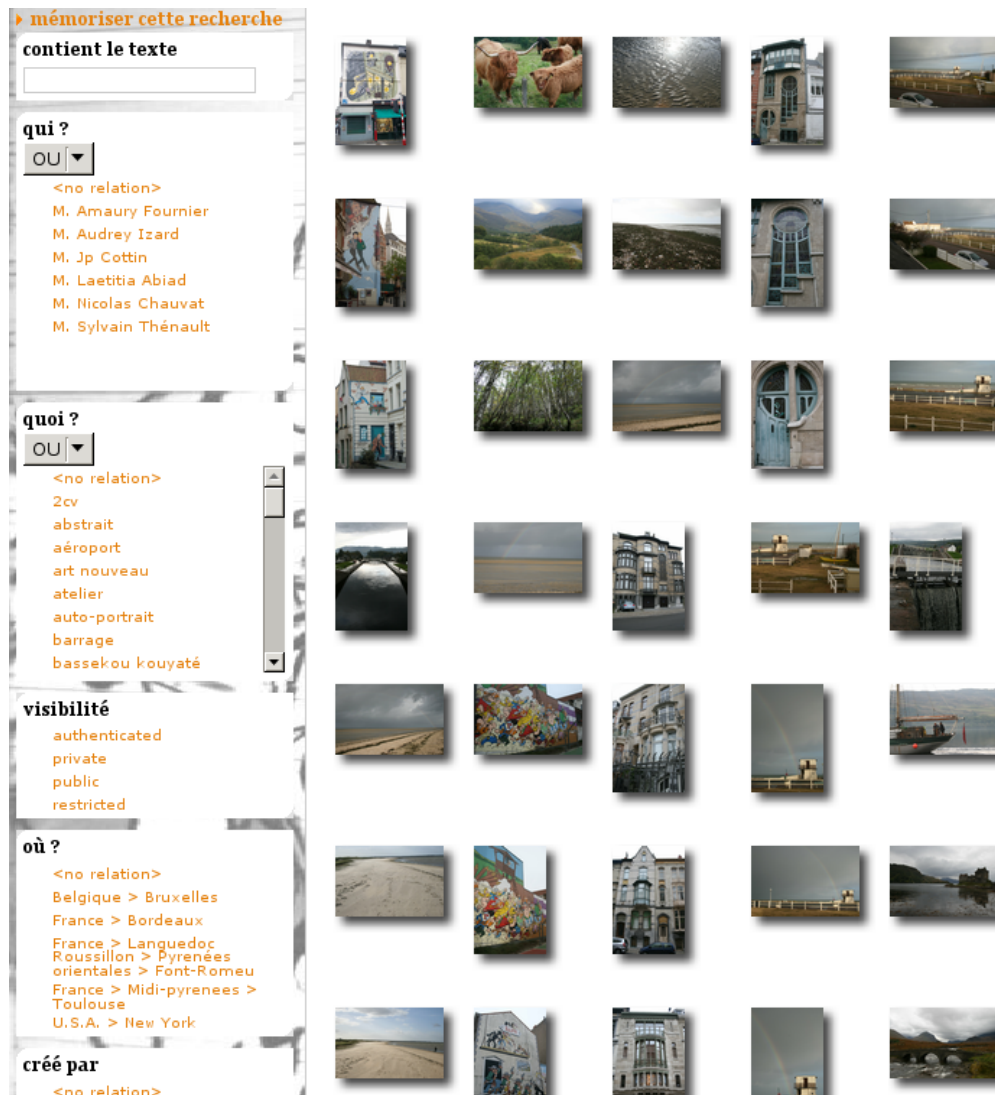
```
from cubicweb.web import facet

class DisplayedOnFacet(facet.RelationFacet):
    __regid__ = 'displayed_on-facet'
    # relation to be displayed
    rtype = 'displayed_on'
    role = 'object'
    # view to use to display persons
    label_vid = 'combobox'
```

Let's say we also want to filter according to the `visibility` attribute. This is even simpler as we just have to derive from the `AttributeFacet` class:

```
class VisibilityFacet(facet.AttributeFacet):
    __regid__ = 'visibility-facet'
    rtype = 'visibility'
```

Now if I search for some pictures on my site, I get the following facets available:



Note: By default a facet must be applicable to every entity in the result set and provide at least two elements of vocabulary to be displayed (for instance you won't see the *created_by* facet if the same user has created all entities). This may explain why you don't see yours...

Conclusion

We started to see the power behind the infrastructure provided by the framework, both on the pure ui (CSS, Javascript) side and on the Python side (high level generic classes for components, including boxes and facets). We now have, with a few lines of code, a full-featured web site with a personalized look.

Of course we'll probably want more as time goes, but we can now concentrate on making good pictures, publishing albums and sharing them with friends...

3.3 Use Windmill with CubicWeb

Windmill implements cross browser testing, in-browser recording and playback, and functionality for fast accurate debugging and test environment integration.

Online features [list](#) is available.

3.3.1 Installation

Windmill

You have to install Windmill manually for now. If you're using Debian, there is no binary package ([yet](#)).

The simplest solution is to use a *setuptools/pip* command (for a clean environment, take a look to the [virtualenv](#) project as well):

```
$ pip install windmill
$ curl -O http://github.com/windmill/windmill/tarball/master
```

However, the Windmill project doesn't release frequently. Our recommendation is to use the last snapshot of the Git repository:

```
$ git clone git://github.com/windmill/windmill.git HEAD
$ cd windmill
$ python3 setup.py develop
```

Install instructions are [available](#).

Be sure to have the windmill module in your PYTHONPATH afterwards:

```
$ python3 -c "import windmill"
```

X dummy

In order to reduce unnecessary system load from your test machines, It's recommended to use X dummy server for testing the Unix web clients, you need a dummy video X driver (as *xserver-xorg-video-dummy* package in Debian) coupled with a light X server as [Xvfb](#).

The dummy driver is a special driver available with the XFree86 DDX. To use the dummy driver, simply substitute it for your normal card driver in the Device section of your *xorg.conf* configuration file. For example, if you normally uses an ati driver, then you will have a Device section with Driver "ati" to let the X server know that you want it to load and use the ati driver; however, for these conformance tests, you would change that line to Driver "dummy" and remove any other ati specific options from the Device section.

From: <http://www.x.org/wiki/XorgTesting>

Then, you can run the X server with the following command

```
$ /usr/bin/X11/Xvfb :1 -ac -screen 0 1280x1024x8 -fbdir /tmp
```

3.3.2 Windmill usage

Record your use case

- start your instance manually
- start **Windmill** with url site as last argument (read **Usage** or use `'-h'` option to find required command line arguments)
- use the record button
- click on save to obtain python code of your use case
- copy the content to a new file in a *windmill* directory

If you are using firefox as client, consider the “firebug” option.

If you have a running instance, you can refine the test by the *loadtest* windmill option:

```
$ windmill -m firebug loadtest=<test_file.py> <instance url>
```

Or use the internal windmill shell to explore available commands:

```
$ windmill -m firebug shell <instance url>
```

And enter python commands:

```
>>> load_test(<your test file>)
>>> run_test(<your test file>)
```

3.3.3 Integrate Windmill tests into CubicWeb

Set environment

You have to create a new unit test file and a *windmill* directory and copy all your windmill use case into it.

```
# test_windmill.py

# Run all scenarii found in windmill directory
from cubicweb.devtools.cwwindmill import (CubicWebWindmillUseCase,
                                           unittest_main)

if __name__ == '__main__':
    unittest_main()
```

Run your tests

You can easily run your windmill test suite through *pytest* or *unittest*. You have to copy a *test_windmill.py* file from `web.test`.

To run your test series:

```
$ pytest test/test_windmill.py
```

By default, CubicWeb will use **firefox** as the default browser and will try to run test instance server on localhost. In the general case, You've no need to change anything.

Check `cubicweb.devtools.cwwindmill.CubicWebWindmillUseCase` for Windmill configuration. You can edit windmill settings with following class attributes:

- browser identification string (firefox|ie|safari|chrome) (firefox by default)
- test_dir testing file path or directory (windmill directory under your unit case file by default)
- edit_test load and edit test for debugging (False by default)

Examples:

```
browser = 'firefox'
test_dir = osp.join(__file__, 'windmill')
edit_test = False
```

If you want to change cubicweb test server parameters, you can check class variables from `CubicWebServerConfig` or inherit it with overriding the `configcls` attribute in `CubicWebServerTC`

```
.. sourcecode:: python
```

```
class OtherCubicWebServerConfig(CubicWebServerConfig): port = 9999

class NewCubicWebServerTC(CubicWebServerTC): configcls = OtherCubicWebServerConfig
```

For instance, CubicWeb framework windmill tests can be manually run by:

```
$ pytest web/test/test_windmill.py
```

Edit your tests

You can toggle the `edit_test` variable to enable test edition.

But if you are using `pytest` as test runner, use the `-i` option directly. The test series will be loaded and you can run assertions step-by-step:

```
$ pytest -i test/test_windmill.py
```

In this case, the *firebug* extension will be loaded automatically for you.

Afterwards, don't forget to save your edited test into the right file (no autosave feature).

Best practises

Don't run another instance on the same port. You risk to silence some regressions (test runner will automatically fail in further versions).

Start your use case by using an assert on the expected primary url page. Otherwise all your tests could fail without clear explanation of the used navigation.

In the same location of the `test_windmill.py`, create a `windmill/` with your windmill recorded use cases.

3.3.4 Caveats

File Upload

Windmill can't do file uploads. This is a limitation of browser Javascript support / sandboxing, not of Windmill per se. It would be nice if there were some command that would prime the Windmill HTTP proxy to add a particular file to the next HTTP request that comes through, so that uploads could at least be faked.

3.3.5 Preferences

A *.windmill/prefs.py* could be used to redefine default configuration values.

For managing browser extensions, read [advanced topic chapter](#).

More configuration examples could be seen in *windmill/conf/global_settings.py* as template.

3.4 Writing text reports with RestructuredText

CubicWeb offers several text formats for the RichString type used in schemas, including restructuredtext.

Three additional restructuredtext roles are defined by *CubicWeb*:

```
cubicweb.ext.rest.eid_reference_role(role, rawtext, text, lineno, inliner, options={}, content=[])
```

```
cubicweb.ext.rest.rql_role(role, rawtext, text, lineno, inliner, options={}, content=[])
```

```
:rql: `<rql-expr>` or :rql: `<rql-expr>:<vid>`
```

Example: `:rql: `Any X,Y WHERE X is CWUser, X login Y:table``

Replace the directive with the output of applying the view to the resultset returned by the query.

“X eid %(userid)s” can be used in the RQL query for this query will be executed with the argument { ‘userid’: `_cw.user.eid` }.

```
cubicweb.ext.rest.bookmark_role(role, rawtext, text, lineno, inliner, options={}, content=[])
```

```
:bookmark: `<bookmark-eid>` or :bookmark: `<eid>:<vid>`
```

Example: `:bookmark: `1234:table``

Replace the directive with the output of applying the view to the resultset returned by the query stored in the bookmark. By default, the view is the one stored in the bookmark, but it can be overridden by the directive as in the example above.

“X eid %(userid)s” can be used in the RQL query stored in the Bookmark, for this query will be executed with the argument { ‘userid’: `_cw.user.eid` }.

3.5 Importing relational data into a CubicWeb instance

3.5.1 Introduction

This tutorial explains how to import data from an external source (e.g. a collection of files) into a CubicWeb cube instance.

First, once we know the format of the data we wish to import, we devise a *data model*, that is, a CubicWeb (Yams) schema which reflects the way the data is structured. This schema is implemented in the `schema.py` file. In this tutorial, we will describe such a schema for a particular data set, the Diseasome data (see below).

Once the schema is defined, we create a cube and an instance. The cube is a specification of an application, whereas an instance is the application per se.

Once the schema is defined and the instance is created, the import can be performed, via the following steps:

1. Build a custom parser for the data to be imported. Thus, one obtains a Python memory representation of the data.
2. Map the parsed data to the data model defined in `schema.py`.
3. Perform the actual import of the data. This comes down to “populating” the data model with the memory representation obtained at 1, according to the mapping defined at 2.

This tutorial illustrates all the above steps in the context of relational data stored in the RDF format.

More specifically, we describe the import of [Diseasome](#) RDF/OWL data.

3.5.2 Building a data model

The first thing to do when using CubicWeb for creating an application from scratch is to devise a *data model*, that is, a relational representation of the problem to be modeled or of the structure of the data to be imported.

In such a schema, we define an entity type (`EntityType` objects) for each type of entity to import. Each such type has several attributes. If the attributes are of known CubicWeb (Yams) types, viz. numbers, strings or characters, then they are defined as attributes, as e.g. `attribute = Int()` for an attribute named `attribute` which is an integer.

Each such type also has a set of relations, which are defined like the attributes, except that they represent, in fact, relations between the entities of the type under discussion and the objects of a type which is specified in the relation definition.

For example, for the Diseasome data, we have two types of entities, genes and diseases. Thus, we create two classes which inherit from `EntityType`:

```
class Disease(EntityType):
    # Corresponds to http://www.w3.org/2000/01/rdf-schema#label
    label = String(maxsize=512, fulltextindexed=True)
    ...

    #Corresponds to http://www4.wiwiiss.fu-berlin.de/diseasome/resource/diseasome/
    ↪ associatedGene
    associated_genes = SubjectRelation('Gene', cardinality='**')
    ...

    #Corresponds to 'http://www4.wiwiiss.fu-berlin.de/diseasome/resource/diseasome/
    ↪ chromosomalLocation'
    chromosomal_location = SubjectRelation('ExternalUri', cardinality='?*', inlined=True)

class Gene(EntityType):
    ...
```

In this schema, there are attributes whose values are numbers or strings. Thus, they are defined by using the CubicWeb / Yams primitive types, e.g., `label = String(maxsize=12)`. These types can have several constraints or attributes, such as `maxsize`. There are also relations, either between the entity types themselves, or between them and a CubicWeb type, `ExternalUri`. The latter defines a class of URI objects in CubicWeb. For instance, the `chromosomal_location`

attribute is a relation between a Disease entity and an ExternalUri entity. The relation is marked by the CubicWeb / Yams SubjectRelation method. The latter can have several optional keyword arguments, such as cardinality which specifies the number of subjects and objects related by the relation type specified. For example, the '?' cardinality in the chromosomal_relation relation type says that zero or more Disease entities are related to zero or one ExternalUri entities. In other words, a Disease entity is related to at most one ExternalUri entity via the chromosomal_location relation type, and that we can have zero or more Disease entities in the data base. For a relation between the entity types themselves, the associated_genes between a Disease entity and a Gene entity is defined, so that any number of Gene entities can be associated to a Disease, and there can be any number of Disease s if a Gene exists.

Of course, before being able to use the CubicWeb / Yams built-in objects, we need to import them:

```
from yams.buildobjs import EntityType, SubjectRelation, String, Int
from cubicweb.schemas.base import ExternalUri
```

3.5.3 Building a custom data parser

The data we wish to import is structured in the RDF format, as a text file containing a set of lines. On each line, there are three fields. The first two fields are URIs (“Universal Resource Identifiers”). The third field is either an URI or a string. Each field bares a particular meaning:

- the leftmost field is an URI that holds the entity to be imported. Note that the entities defined in the data model (i.e., in schema.py) should correspond to the entities whose URIs are specified in the import file.
- the middle field is an URI that holds a relation whose subject is the entity defined by the leftmost field. Note that this should also correspond to the definitions in the data model.
- the rightmost field is either an URI or a string. When this field is an URI, it gives the object of the relation defined by the middle field. When the rightmost field is a string, the middle field is interpreted as an attribute of the subject (introduced by the leftmost field) and the rightmost field is interpreted as the value of the attribute.

Note however that some attributes (i.e. relations whose objects are strings) have their objects defined as strings followed by ^^ and by another URI; we ignore this part.

Let us show some examples:

- of line holding an attribute definition: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/genes/CYP17A1> <http://www.w3.org/2000/01/rdf-schema#label> "CYP17A1" . The line contains the definition of the label attribute of an entity of type gene. The value of label is 'CYP17A1'.
- of line holding a relation definition: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseases/1> <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseasome/associatedGene> <http://www4.wiwiss.fu-berlin.de/diseasome/resource/genes/HADH2> . The line contains the definition of the associatedGene relation between a disease subject entity identified by 1 and a gene object entity defined by HADH2.

Thus, for parsing the data, we can (note: see the diseasome_parser module):

1. define a couple of regular expressions for parsing the two kinds of lines, RE_ATTIS for parsing the attribute definitions, and RE_RELS for parsing the relation definitions.
2. define a function that iterates through the lines of the file and retrieves (yield s) a (subject, relation, object) tuple for each line. We called it _retrieve_structure in the diseasome_parser module. The function needs the file name and the types for which information should be retrieved.

Alternatively, instead of hand-making the parser, one could use the RDF parser provided in the dataio cube.

Once we get to have the (subject, relation, object) triples, we need to map them into the data model.

3.5.4 Mapping the data to the schema

In the case of diseasome data, we can just define two dictionaries for mapping the names of the relations as extracted by the parser, to the names of the relations as defined in the `schema.py` data model. In the `diseasome_parser` module they are called `MAPPING_ATTS` and `MAPPING_RELS`. Given that the relation and attribute names are given in CamelCase in the original data, mappings are necessary if we follow the PEP08 when naming the attributes in the data model. For example, the RDF relation `chromosomalLocation` is mapped into the schema relation `chromosomal_location`.

Once these mappings have been defined, we just iterate over the (subject, relation, object) tuples provided by the parser and we extract the entities, with their attributes and relations. For each entity, we thus have a dictionary with two keys, `attributes` and `relations`. The value associated to the `attributes` key is a dictionary containing (attribute: value) pairs, where “value” is a string, plus the `cwuri` key / attribute holding the URI of the entity itself. The value associated to the `relations` key is a dictionary containing (relation: value) pairs, where “value” is an URI. This is implemented in the `entities_from_rdf` interface function of the module `diseasome_parser`. This function provides an iterator on the dictionaries containing the `attributes` and `relations` keys for all entities.

However, this is a simple case. In real life, things can get much more complicated, and the mapping can be far from trivial, especially when several data sources (which can follow different formatting and even structuring conventions) must be mapped into the same data model.

3.5.5 Importing the data

The data import code should be placed in a Python module. Let us call it `diseasome_import.py`. Then, this module should be called via `cubicweb-ctl`, as follows:

```
cubicweb-ctl shell diseasome_import.py -- <other arguments e.g. data file>
```

In the import module, we should use a *store* for doing the import. A store is an object which provides three kinds of methods for importing data:

- a method for importing the entities, along with the values of their attributes.
- a method for importing the relations between the entities.
- a method for committing the imports to the database.

In CubicWeb, we have four stores:

1. `ObjectStore` base class for the stores in CubicWeb. It only provides a skeleton for all other stores and provides the means for creating the memory structures (dictionaries) that hold the entities and the relations between them.
2. `RQLObjectStore`: store which uses the RQL language for performing database insertions and updates. It relies on all the CubicWeb hooks machinery, especially for dealing with security issues (database access permissions).
2. `NoHookRQLObjectStore`: store which uses the RQL language for performing database insertions and updates, but for which all hooks are deactivated. This implies that certain checks with respect to the CubicWeb / Yams schema (data model) are not performed. However, all SQL queries obtained from the RQL ones are executed in a sequential manner, one query per inserted entity.
4. `SQLGenObjectStore`: store which uses the SQL language directly. It inserts entities either sequentially, by executing an SQL query for each entity, or directly by using one PostGRES `COPY FROM` query for a set of similarly structured entities.

For really massive imports (millions or billions of entities), there is a `cube dataio` which contains another store, called `MassiveObjectStore`. This store is similar to `SQLGenObjectStore`, except that anything related to CubicWeb is bypassed. That is, even the CubicWeb EID entity identifiers are not handled. This store is the fastest, but has a slightly

different API from the other four stores mentioned above. Moreover, it has an important limitation, in that it doesn't insert inlined¹ relations in the database.

In the following section we will see how to import data by using the stores in CubicWeb's `dataimport` module.

Using the stores in `dataimport`

`ObjectStore` is seldom used in real life for importing data, since it is only the base store for the other stores and it doesn't perform an actual import of the data. Nevertheless, the other three stores, which import data, are based on `ObjectStore` and provide the same API.

All three stores `RQLObjectStore`, `NoHookRQLObjectStore` and `SQLGenObjectStore` provide exactly the same API for importing data, that is entities and relations, in an SQL database.

Before using a store, one must import the `dataimport` module and then initialize the store, with the current session as a parameter:

```
import cubicweb.dataimport as cwdi
...

store = cwdi.RQLObjectStore(session)
```

Each such store provides three methods for data import:

1. `create_entity(Etype, **attributes)`, which allows us to add an entity of the Yams type `Etype` to the database. This entity's attributes are specified in the `attributes` dictionary. The method returns the entity created in the database. For example, we add two entities, a person, of `Person` type, and a location, of `Location` type:

```
person = store.create_entity('Person', name='Toto', age='18', height='190')

location = store.create_entity('Location', town='Paris', arrondissement='13')
```

2. `relate(subject_eid, r_type, object_eid)`, which allows us to add a relation of the Yams type `r_type` to the database. The relation's subject is an entity whose EID is `subject_eid`; its object is another entity, whose EID is `object_eid`. For example²:

```
store.relate(person.eid(), 'lives_in', location.eid(), **kwargs)
```

`kwargs` is only used by the `SQLGenObjectStore`'s `relate` method and is here to allow us to specify the type of the subject of the relation, when the relation is defined as inlined in the schema.

1. `flush()`, which allows us to perform the actual commit into the database, along with some cleanup operations. Ideally, this method should be called as often as possible, that is after each insertion in the database, so that

¹ An inlined relation is a relation defined in the schema with the keyword argument `inlined=True`. Such a relation is inserted in the database as an attribute of the entity whose subject it is.

² **The `eid` method of an entity defined via `create_entity` returns** the entity identifier as assigned by CubicWeb when creating the entity. This only works for entities defined via the stores in the CubicWeb's `dataimport` module.

The keyword argument that is understood by `SQLGenObjectStore` is called `subtype` and holds the type of the subject entity. For the example considered here, this comes to having³:

```
store.relate(person.eid(), 'lives_in', location.eid(), subtype=person.cw_etype)
```

If `subtype` is not specified, then the store tries to infer the type of the subject. However, this doesn't always work, e.g. when there are several possible subject types for a given relation type.

³ **The `cw_etype` attribute of an entity defined via `create_entity` holds** the type of the entity just created. This only works for entities defined via the stores in the CubicWeb's `dataimport` module. In the example considered here, `person.cw_etype` holds `'Person'`. All the other stores but `SQLGenObjectStore` ignore the `kwargs` parameters.

database sessions are kept as atomic as possible. In practice, we usually call this method twice: first, after all the entities have been created, second, after all relations have been created.

Note however that before each commit the database insertions have to be consistent with the schema. Thus, if, for instance, an entity has an attribute defined through a relation (viz. a `SubjectRelation`) with a "1" or "+" object cardinality, we have to create the entity under discussion, the object entity of the relation under discussion, and the relation itself, before committing the additions to the database.

The `flush` method is simply called as:

```
store.flush()
```

Using the `MassiveObjectStore` in the `dataio` cube

This store, available in the `dataio` cube, allows us to fully dispense with the CubicWeb import mechanisms and hence to interact directly with the database server, via SQL queries.

Moreover, these queries rely on PostgreSQL's `COPY FROM` instruction to create several entities in a single query. This brings tremendous performance improvements with respect to the RQL-based data insertion procedures.

However, the API of this store is slightly different from the API of the stores in CubicWeb's `dataimport` module.

Before using the store, one has to import the `dataio` cube's `dataimport` module, then initialize the store by giving it the `session` parameter:

```
from cubicweb_dataio import dataimport as mcwdi
...

store = mcwdi.MassiveObjectStore(session)
```

The `MassiveObjectStore` provides six methods for inserting data into the database:

1. `init_rtype_table(ObjEtype, r_type, ObjEtype)`, which specifies the creation of the tables associated to the relation types in the database. Each such table has three columns, the type of the subject entity, the type of the relation (that is, the name of the attribute in the subject entity which is defined via the relation), and the type of the object entity. For example:

```
store.init_rtype_table('Person', 'lives_in', 'Location')
```

Please note that these tables can be created before the entities, since they only specify their types, not their unique identifiers.

2. `create_entity(Etype, **attributes)`, which allows us to add new entities, whose attributes are given in the `attributes` dictionary. Please note however that, by default, this method does *not* return the created entity. The method is called, for example, as in:

```
store.create_entity('Person', name='Toto', age='18', height='190',
                    uri='http://link/to/person/toto_18_190')
store.create_entity('Location', town='Paris', arrondissement='13',
                    uri='http://link/to/location/paris_13')
```

In order to be able to link these entities via the relations when needed, we must provide ourselves a means for uniquely identifying the entities. In general, this is done via URIs, stored in attributes like `uri` or `cwuri`. The name of the attribute is irrelevant as long as its value is unique for each entity.

3. `relate_by_iid(subject_iid, r_type, object_iid)` allows us to actually relate the entities uniquely identified by `subject_iid` and `object_iid` via a relation of type `r_type`. For example:

```
store.relate_by_iid('http://link/to/person/toto_18_190',
                   'lives_in',
                   'http://link/to/location/paris_13')
```

Please note that this method does *not* work for inlined relations!

4. `convert_relations(ObjEtype, r_type, ObjEtype, subj_iid_attribute, obj_iid_attribute)` allows us to actually insert the relations in the database. At one call of this method, one inserts all the relations of type `r_type` between entities of given types. `subj_iid_attribute` and `object_iid_attribute` are the names of the attributes which store the unique identifiers of the entities, as assigned by the user. These names can be identical, as long as their values are unique. For example, for inserting all relations of type `lives_in` between `People` and `Location` entities, we write:

```
store.convert_relations('Person', 'lives_in', 'Location', 'uri', 'uri')
```

5. `flush()` performs the actual commit in the database. It only needs to be called after `create_entity` and `relate_by_iid` calls. Please note that `relate_by_iid` does *not* perform insertions into the database, hence calling `flush()` for it would have no effect.
6. `cleanup()` performs database cleanups, by removing temporary tables. It should only be called at the end of the import.

Application to the Diseasome data

Import setup

We define an import function, `diseasome_import`, which does basically four things:

1. creates and initializes the store to be used, via a line such as:

```
store = cwdi.SQLGenObjectStore(session)
```

where `cwdi` is the imported `cubicweb.dataimport` or `cubicweb_dataio.dataimport`.

2. calls the diseasome parser, that is, the `entities_from_rdf` function in the `diseasome_parser` module and iterates on its result, in a line such as:

```
for entity, relations in parser.entities_from_rdf(filename, ('gene', 'disease')):
```

where `parser` is the imported `diseasome_parser` module, and `filename` is the name of the file containing the data (with its path), e.g. `../data/diseasome_dump.nt`.

3. creates the entities to be inserted in the database; for Diseasome, there are two kinds of entities:

1. entities defined in the data model, viz. `Gene` and `Disease` in our case.
2. entities which are built in CubicWeb / Yams, viz. `ExternalUri` which define URIs.

As we are working with RDF data, each entity is defined through a series of URIs. Hence, each “relational attribute”⁴ of an entity is defined via an URI, that is, in CubicWeb terms, via an `ExternalUri` entity. The entities are created, in the loop presented above, as such:

⁴

By “relational attribute” we denote an attribute (of an entity) which is defined through a relation, e.g. the `chromosomal_location` attribute of `Disease` entities, which is defined through a relation between a `Disease` and an `ExternalUri`.

The `ExternalUri` entities are as many as URIs in the data file. For them, we define a unique attribute, `uri`, which holds the URI under discussion:

```
extu = store.create_entity('ExternalUri', uri="http://path/of/the/uri")
```

```
ent = store.create_entity(etype, **entity)
```

where `etype` is the appropriate entity type, either `Gene` or `Disease`.

1. creates the relations between the entities. We have relations between:

1. entities defined in the schema, e.g. between `Disease` and `Gene` entities, such as the `associated_genes` relation defined for `Disease` entities.
2. entities defined in the schema and `ExternalUri` entities, such as `gene_id`.

The way relations are added to the database depends on the store:

- for the stores in the CubicWeb `dataimport` module, we only use `store.relate`, in another loop, on the relations (that is, a loop inside the preceding one, mentioned at step 2):

```
for rtype, rels in relations.iteritems():
    ...

    store.relate(ent.eid(), rtype, extu.eid(), **kwargs)
```

where `kwargs` is a dictionary designed to accommodate the need for specifying the type of the subject entity of the relation, when the relation is inlined and `SQLGenObjectStore` is used. For example:

```
...
store.relate(ent.eid(), 'chromosomal_location', extu.eid(), subtype='Disease')
```

- for the `MassiveObjectStore` in the `dataio` cube's `dataimport` module, the relations are created in three steps:
 1. first, a table is created for each relation type, as in:

```
...
store.init_rtype_table(ent.cw_etype, rtype, extu.cw_etype)
```

which comes down to lines such as:

```
store.init_rtype_table('Disease', 'associated_genes', 'Gene')
store.init_rtype_table('Gene', 'gene_id', 'ExternalUri')
```

2. second, the URI of each entity will be used as its identifier, in the `relate_by_iid` method, such as:

```
disease_uri = 'http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseases/3
↳ '
gene_uri = '<http://www4.wiwiss.fu-berlin.de/diseasome/resource/genes/HSD3B2
↳ '
store.relate_by_iid(disease_uri, 'associated_genes', gene_uri)
```

3. third, the relations for each relation type will be added to the database, via the `convert_relations` method, such as in:

```
store.convert_relations('Disease', 'associated_genes', 'Gene', 'cwuri',
↳ 'cwuri')
```

and:

```
store.convert_relations('Gene', 'hgnc_id', 'ExternalUri', 'cwuri', 'uri')
```

where `cwuri` and `uri` are the attributes which store the URIs of the entities defined in the data model, and of the `ExternalUri` entities, respectively.

2. flushes all relations and entities:

```
store.flush()
```

which performs the actual commit of the inserted entities and relations in the database.

If the `MassiveObjectStore` is used, then a cleanup of temporary SQL tables should be performed at the end of the import:

```
store.cleanup()
```

Timing benchmarks

In order to time the import script, we just decorate the import function with the `timed` decorator:

```
from logilab.common.decorators import timed
...

@timed
def diseasome_import(session, filename):
    ...
```

After running the import function as shown in the “Importing the data” section, we obtain two time measurements:

```
diseasome_import clock: ... / time: ...
```

Here, the meanings of these measurements are⁵:

- `clock` is the time spent by CubicWeb, on the server side (i.e. hooks and data pre- / post-processing on SQL queries),
- `time` is the sum between `clock` and the time spent in PostgreSQL.

The import function is put in an import module, named `diseasome_import` here. The module is called directly from the CubicWeb shell, as follows:

```
cubicweb-ctl shell diseasome_instance diseasome_import.py \
-- -df diseasome_import_file.nt -st StoreName
```

The module accepts two arguments:

- the data file, introduced by `-df [--datafile]`, and
- the store, introduced by `-st [--store]`.

The timings (in seconds) for different stores are given in the following table, for importing 4213 Disease entities and 3919 Gene entities with the import module just described:

⁵ The meanings of the `clock` and `time` measurements, when using the `@timed` decorators, were taken from [a blog post on massive data import in CubicWeb](#).

Store	CubicWeb time (clock)	PostgreSQL time (time - clock)	Total time
RQLObjectStore	225.98	62.05	288.03
NoHookRQLObjectStore	62.73	51.38	114.11
SQLGenObjectStore	20.41	11.03	31.44
MassiveObjectStore	4.84	6.93	11.77

3.5.6 Conclusions

In this tutorial we have seen how to import data in a CubicWeb application instance. We have first seen how to create a schema, then how to create a parser of the data and a mapping of the data to the schema. Finally, we have seen four ways of importing data into CubicWeb.

Three of those are integrated into CubicWeb, namely the RQLObjectStore, NoHookRQLObjectStore and SQLGenObjectStore stores, which have a common API:

- RQLObjectStore is by far the slowest, especially its time spent on the CubicWeb side, and so it should be used only for small amounts of “sensitive” data (i.e. where security is a concern).
- NoHookRQLObjectStore slashes by almost four the time spent on the CubicWeb side, but is also quite slow; on the PostgreSQL side it is as slow as the previous store. It should be used for data where security is not a concern, but consistency (with the data model) is.
- SQLGenObjectStore slashes by three the time spent on the CubicWeb side and by five the time spent on the PostgreSQL side. It should be used for relatively great amounts of data, where security and data consistency are not a concern. Compared to the previous store, it has the disadvantage that, for inlined relations, we must specify their subjects’ types.

For really huge amounts of data there is a fourth store, MassiveObjectStore, available from the dataio cube. It provides a blazing performance with respect to all other stores: it is almost 25 times faster than RQLObjectStore and almost three times faster than SQLGenObjectStore. However, it has a few usage caveats that should be taken into account:

1. it cannot insert relations defined as inlined in the schema,
2. no security or consistency check is performed on the data,
3. its API is slightly different from the other stores.

Hence, this store should be used when security and data consistency are not a concern, and there are no inlined relations in the schema.

3.6 Create a Website from scratch with CubicWeb

3.6.1 Introduction

This tutorial aims to demonstrate how to create a website using CubicWeb. This website will present museums from French Ministry of Culture data, available [here](#).

First, we will start with installation and creation of our website, and a short presentation of out of the box CubicWeb functionalities. Then, we will see how to enhance our views using Jinja2 templates or React components to have a better looking site. Finally, we will see how to manage more data, and how to serialize them in RDF.

At the end of this tutorial, you will have a website giving information about all France’s museums, describes them in RDF and present them on a map.

You can find the code of the finished tutorial in our forge, look for the cube [tuto](#).

Getting started

Installation of CubicWeb and dependencies

In this tutorial, we choose to install CubicWeb as a Python Package in a Python3 virtual environment, with `pip`; instead of using Debian installation. We will also need to install `psycopg2-binary` if we use a postgresql database:

```
python3 -m venv venv-tuto
source venv-tuto/bin/activate
pip install cubicweb
pip install psycopg2-binary
```

Create a cube

Now we have CubicWeb installed, we will need to create a cube, which will contain our application. We will call our cube `tuto`:

```
cubicweb-ctl newcube tuto -d tuto
```

This command will lead to several questions, as a short description of the new cube. Then, it will create a directory named `tuto` (as we specified it with `-d`) reflecting the structure described in [Standard structure for a cube](#).

To install our new cube on the virtual environment, run in the `tuto/cubicweb-tuto` directory:

```
pip install -e .
```

All `cubicweb-ctl` commands are described in details in [cubicweb-ctl tool](#).

Create and start our instance

Now we created our cube, we need to instantiate it to launch our website:

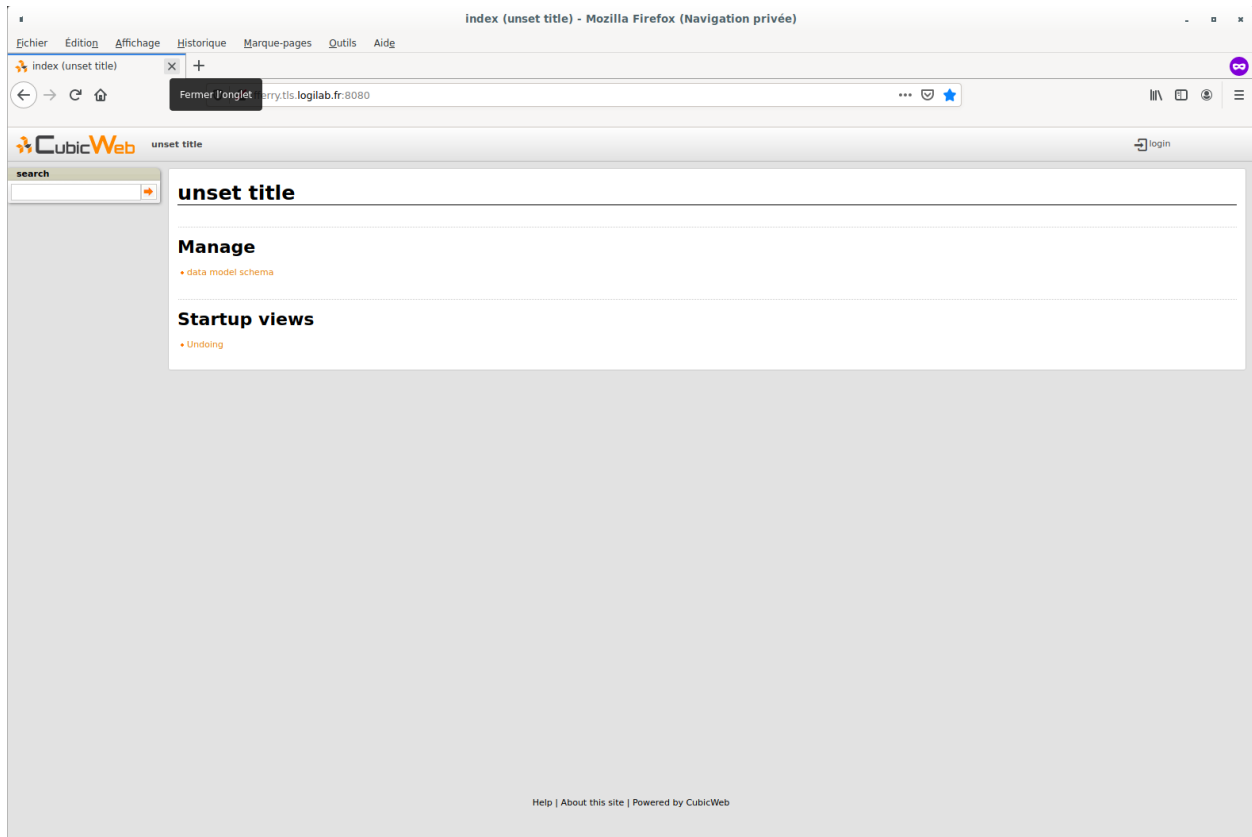
```
cubicweb-ctl create tuto tuto_instance
```

Several questions will be asked to parameter our new instance, most of them can be answered with default value, some (as DB user and password) can't. At the end, it will be asked if we want to create the database now. We do not need it right now, as we will create in further steps.

Then, we can launch our instance in debug mod (with `-D` option):

```
cubicweb-ctl pyramid -D tuto_instance
```

You can now access the instance from <http://localhost:8080>



As you can see, we already have several functionalities which come out-of-the-box, for instance user management, data model schema displaying, etc.

Now, we need to design our data model, to be able to create and display some museums.

Defining our data model

We want to display some museums, each have a name, a postal address, maybe one or several director, a geographical position (latitude and longitude) and are in a city. Some of these concepts will be classes, others attributes.

Thus, we will write the following code in our `tuto/cubicweb_tuto/schema.py` file:

```
from yams.buildobjs import EntityType, String, Float, RelationDefinition, Int

class Museum(EntityType):
    name = String()
    latitude = Float()
    longitude = Float()
    postal_address = String()

class City(EntityType):
    name = String()
    zip_code = Int()
```

(continues on next page)

(continued from previous page)

```

class Person(EntityType):
    name = String()
    email = String()

class is_in(RelationDefinition):
    subject = 'Museum'
    object = 'City'
    cardinality = '1*'

class director(RelationDefinition):
    subject = 'Museum'
    object = 'Person'
    cardinality = '**'

```

The first step is the import from the yams package necessary classes to build the schema.

This file defines the following:

- a *Museum* has a name, a latitude, a longitude and a postal address as attributes.
 - the name and postal address are strings;
 - the latitude and longitude are floating numbers.
- a *City* has a name and a zip code as attributes.
- a *Person* has a name and an email as attributes
- a *Museum* must be linked to a *City* using the *is_in* relation
 - * means a *City* may be linked to 0 to N *Museum*, 1 means a *Museum* must be linked to one and only one *City*. For completeness, you can also use + for 1 to N, and ? for 0 or 1.
- a *Museum* can be linked to 0 or several *Person* using the *director* relation, and a *Person* can be linked to 0 or several *Museum*.

Of course, there are a lot of other data types and things such as constraints, permissions, etc, that may be defined in the schema, but those will not be covered in this tutorial.

In our case, our relations have only on subject type. Thus, we can define them directly in *Museum* class, using *SubjectRelation*, like this:

```

from yams.buildobjs import EntityType, String, Float, SubjectRelation, Int

class Museum(EntityType):
    name = String()
    latitude = Float()
    longitude = Float()
    is_in = SubjectRelation("City", cardinality="1*")
    director = SubjectRelation("Person", cardinality="**")
    postal_address = String()

class City(EntityType):
    name = String()
    zip_code = Int()

```

(continues on next page)

(continued from previous page)

```
class Person(EntityType):
    name = String()
    email = String()
```

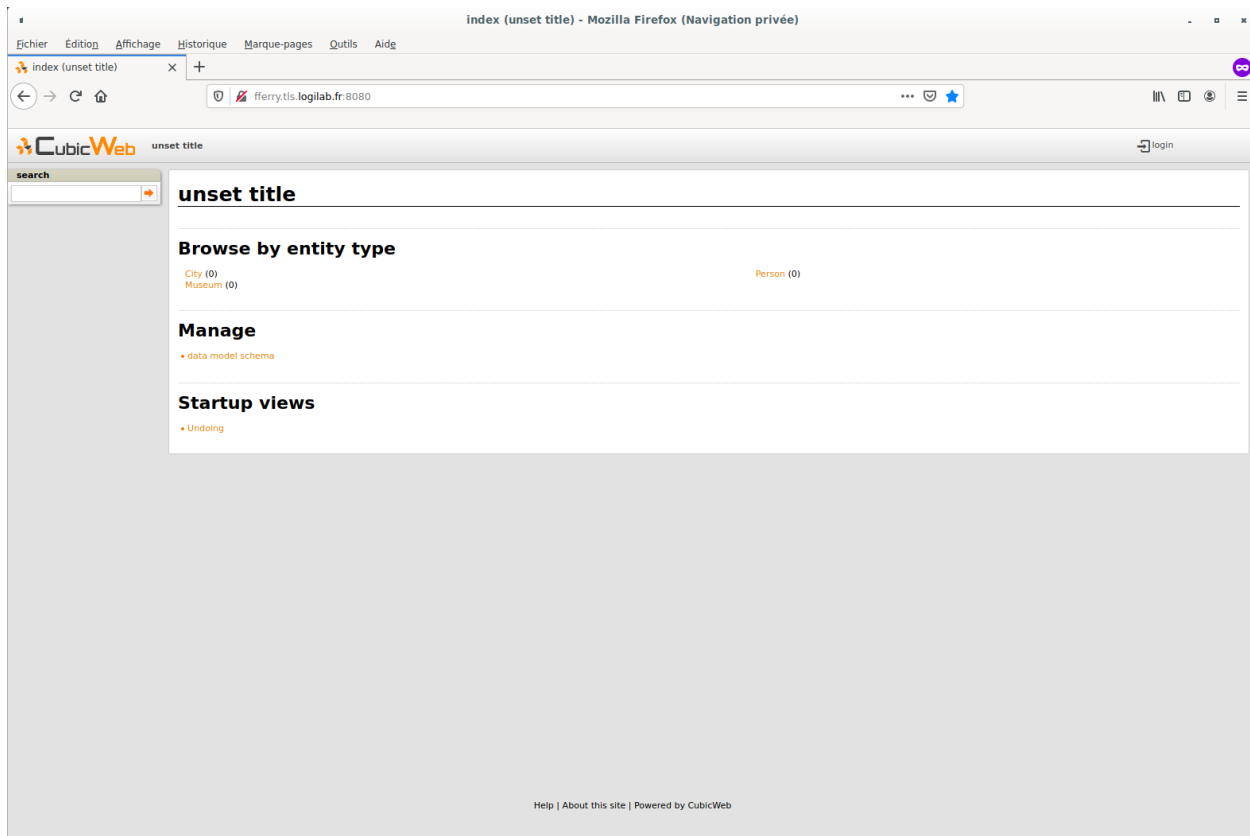
Once the schema created, we need to create our database, and then initialise it with the aforementioned schema:

```
cubicweb-ctl db-create tuto_instance
cubicweb-ctl db-init tuto_instance
```

Note: At the end of the *db-create*, it is asked if we want to run database initialisation, thus we can avoid running *db-init*.

Note: In our case, we had no existing database, so we had to initialize a database. But with a pre-existing database and schema, we have to use migration scripts, see [Migration](#) for more information about this topic.

If we launch again our instance, we should see our new entity types in the homepage: City, Museum, Person; and for each, the number of instance of these types (currently 0, as we don't have any of these entities).

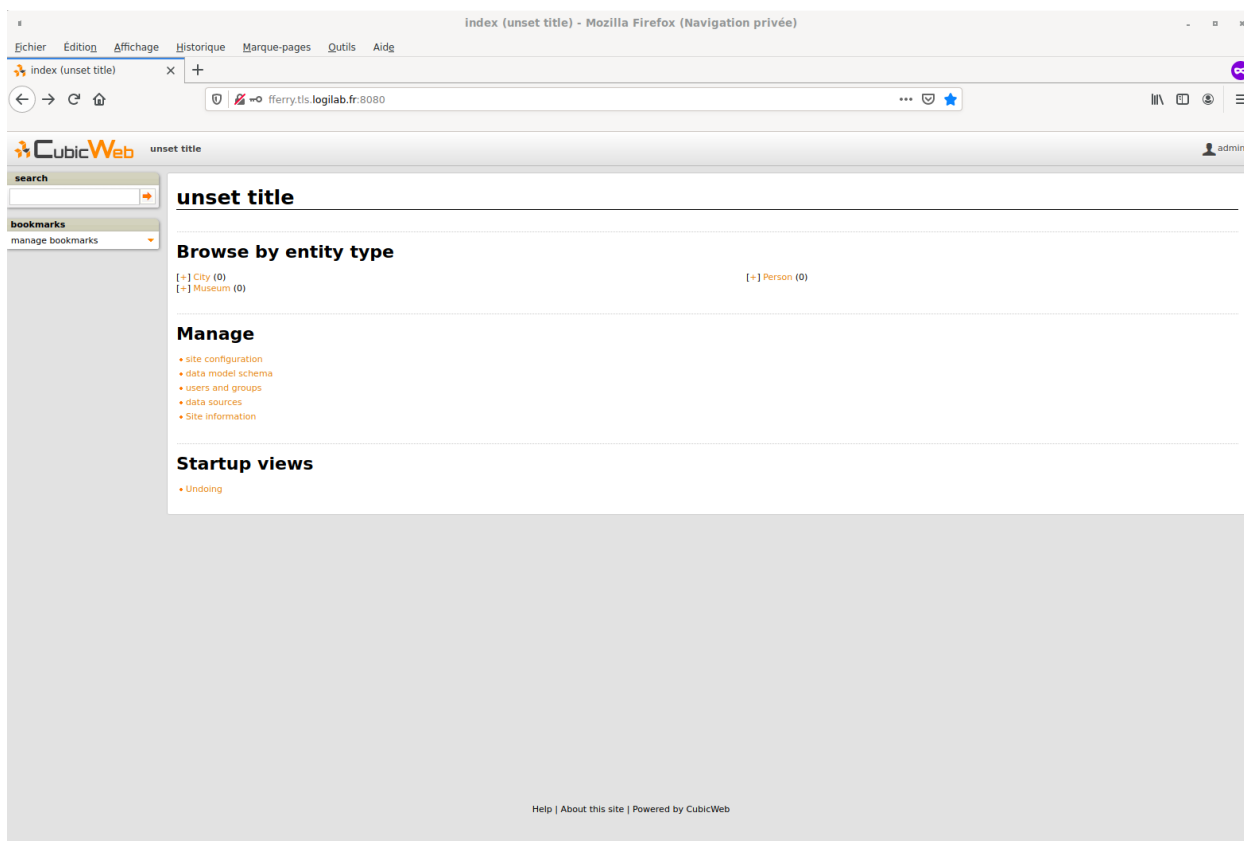


By clicking on *data model schema*, we can see our data model, with our three classes and two relations.

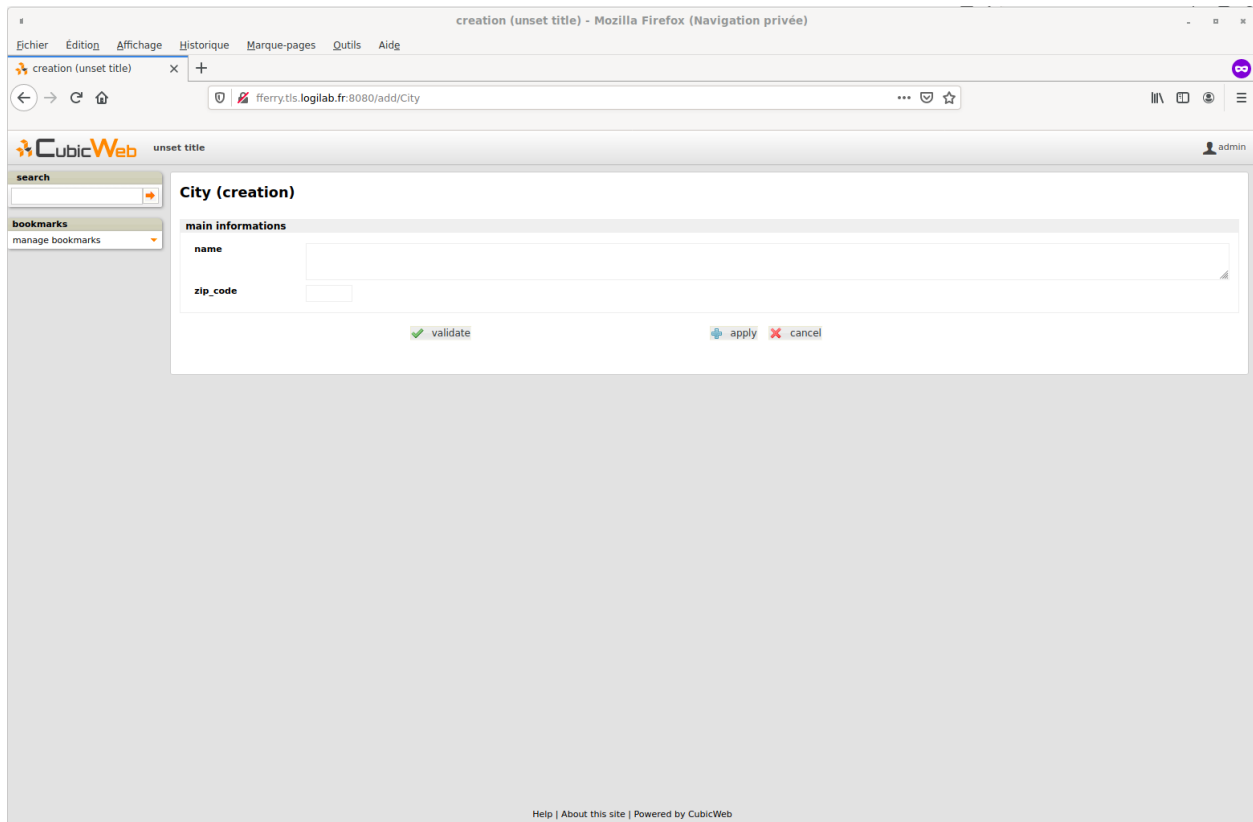
The screenshot shows a web browser window with the title "data model schema (unset title) - Mozilla Firefox (Navigation privée)". The address bar shows the URL "fferrytts.logilab.fr:8080/schema". The CubicWeb logo is in the top left, and a "login" button is in the top right. Below the logo is a search bar. The main content area is titled "data model schema" and has three tabs: "diagram", "entities", and "relations". The "diagram" tab is selected, showing a UML class diagram. The diagram includes three classes: "City", "Person", and "Museum". "City" has a "name (String)" attribute. "Person" has "name (String)" and "email (String)" attributes. "Museum" has "name (String)", "latitude (Float)", "longitude (Float)", and "postal_address (String)" attributes. There is a directed association from "City" to "Museum" labeled "is_in" with multiplicity "0..1" at the "City" end and "0..n" at the "Museum" end. There is a directed association from "Person" to "Museum" labeled "director" with multiplicity "0..1" at the "Person" end and "0..n" at the "Museum" end. Below the diagram, there is a note: "This schema of the data model excludes the meta-data, but you can also display a complete schema with meta-data. Download schema as OWL." At the bottom of the page, there is a footer: "Help | About this site | Powered by CubicWeb".

Adding data

Now we have our entity types defined, we will see how to add some entities. To do this, we need to be connected as administrator, using the *login* button at right top of the site, or visiting <http://localhost:8080/login>. As you can see, we have more choices in the homepage, and beside each entity type, we have a **+**, allowing to create a new entity of this type.

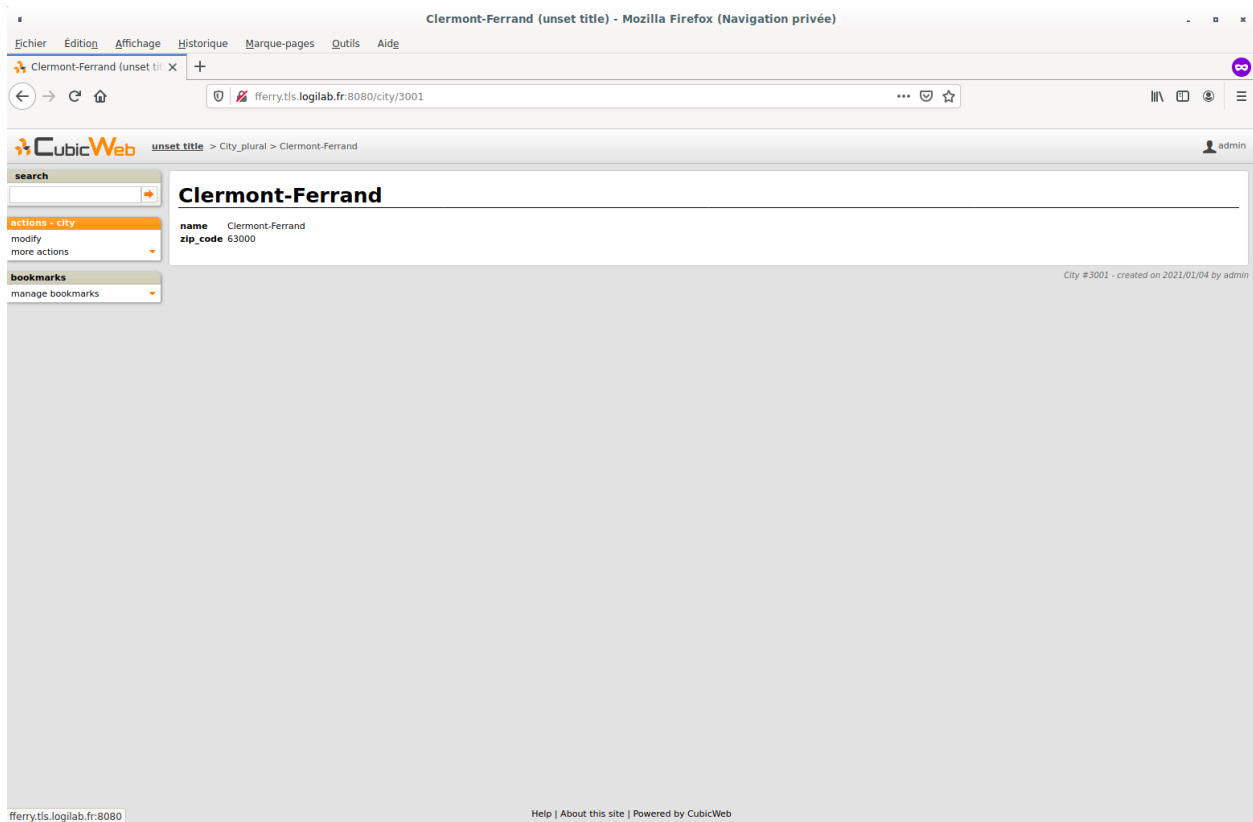


As we built our schema, a Museum have to be linked to a City, so we first need to create a City before adding a museum. To do this, we just have to click on the + beside *City (0)*, and fill the form.

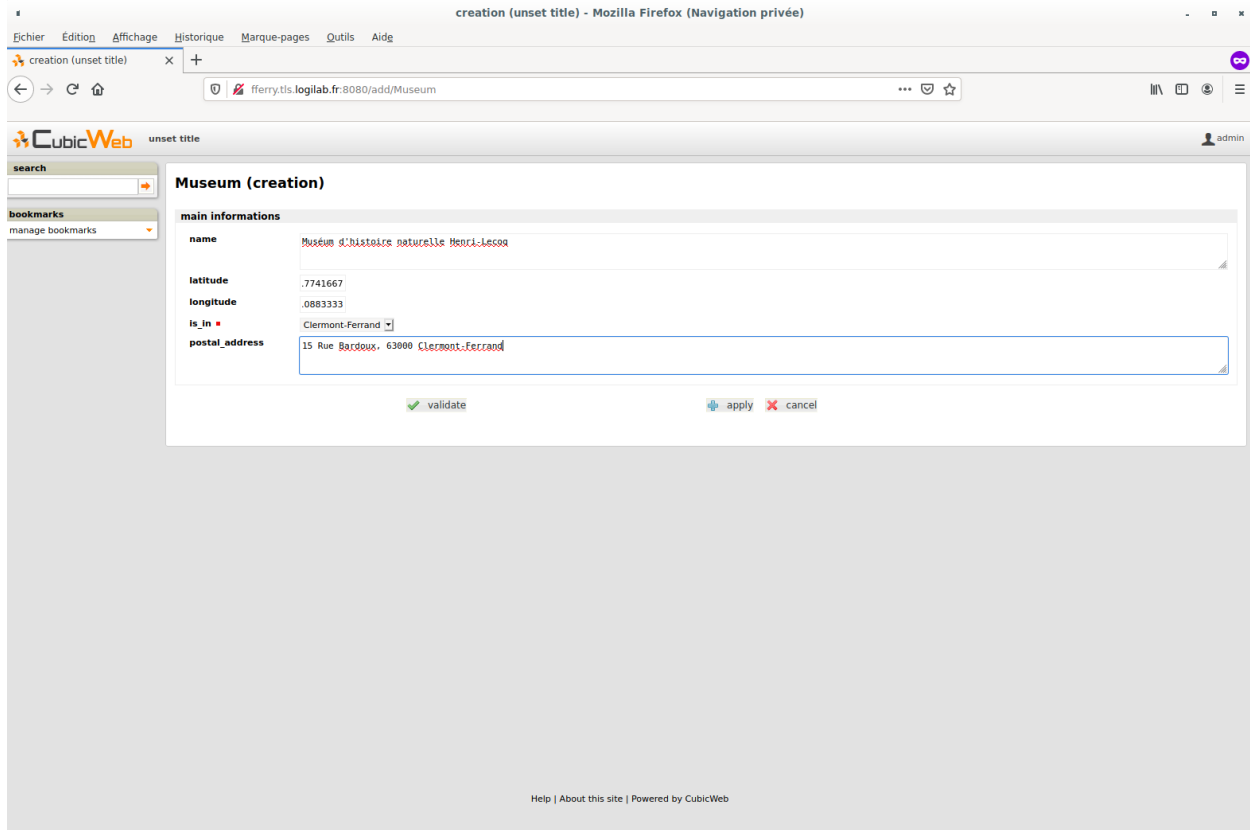


As you can see, all the fields comes directly from the schema and the form is automatically generated by CubicWeb.

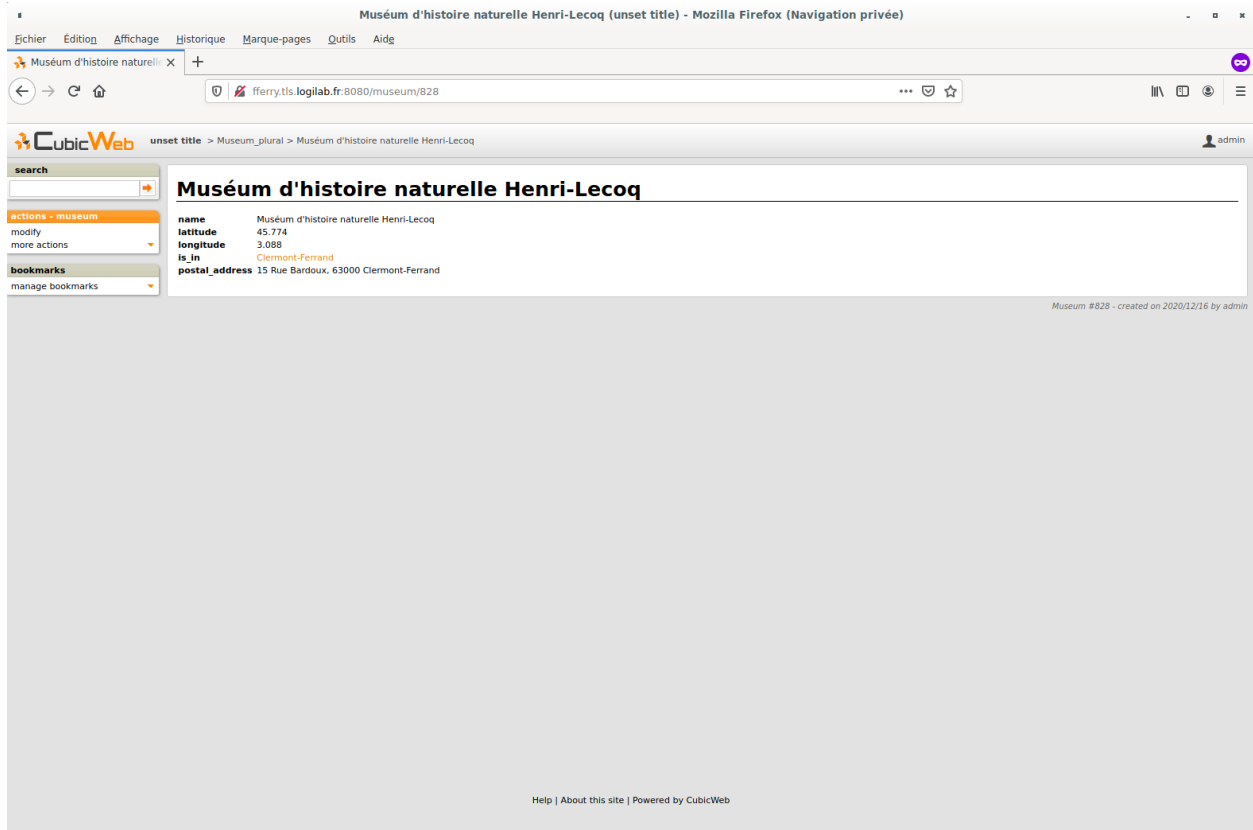
When all the fields are fulfilled, we just have to validate, and we are redirected on the city page, where we can see its different attributes, and on the left, several possible action; as modify our entity or delete it.



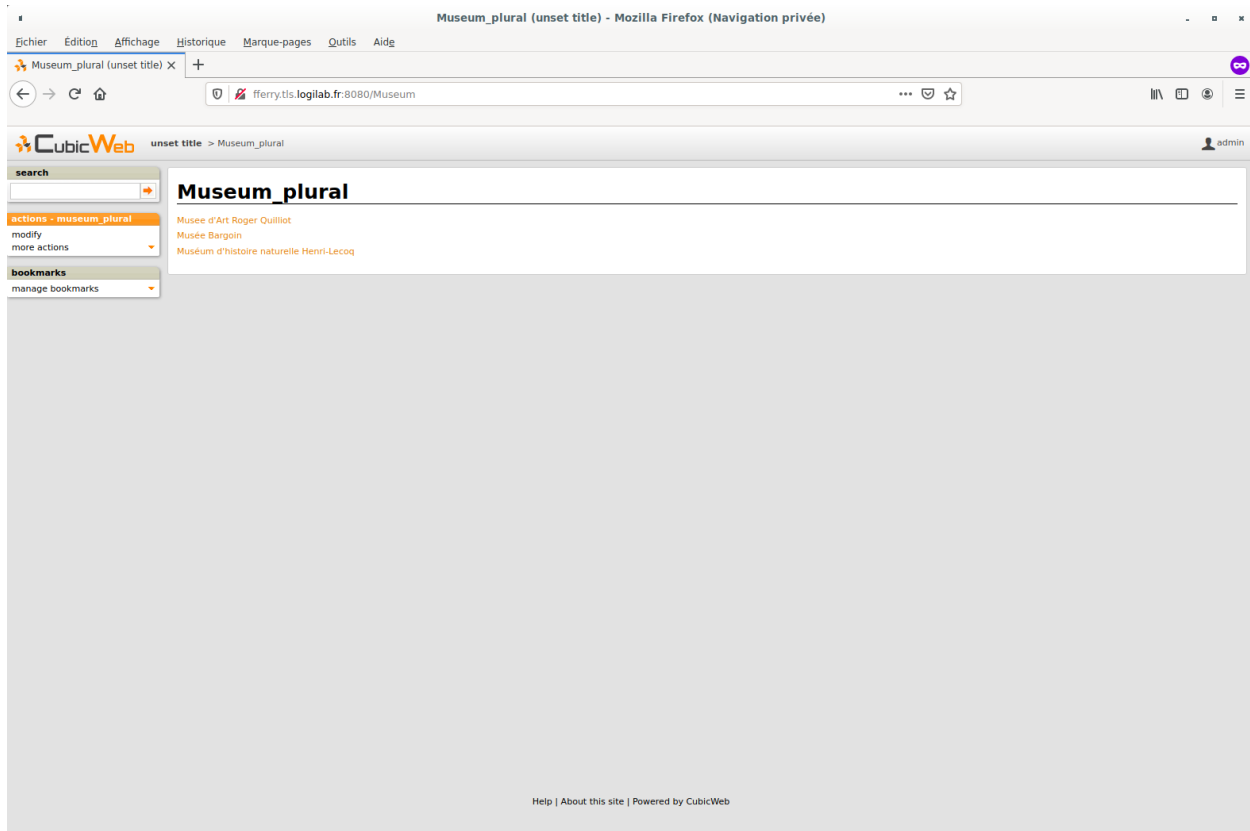
Now we have our first city, we will add its three museums. As for the city creation, we have an autogenerated form; but with a little particularity: a field to choose the city to link with our museum. This field must be fulfilled to create our entity.



As for the city, we are redirected on the entity view after its creation.



We then add two other museums. When we go back to the homepage, we can see all three museums when we click on *Museum_plural* (3).



If we click on *City* in the homepage, we do not have a list view, but our single entity view. This is because in the first case, the framework chose to use the ‘primary’ view since there is only one entity in the data to be displayed. As we have three museums, the ‘list’ view is more appropriate and hence is being used.

There are various other places where *CubicWeb* adapts to display data in the best way, the main being provided by the view *selection* mechanism that will be detailed later.

Customize museum primary view

The ‘primary’ view (i.e. any view with the identifier set to ‘primary’) is the one used to display all the information about a single entity. The standard primary view is one of the most sophisticated views of all. It has several customisation points, but its power comes with *uicfg*, allowing you to control it without having to subclass it. More information are available here : [The Primary View](#).

Now we have several museums, we want an easier way to identify its city when we are on the museum page. To achieve this, we will subclass *PrimaryView* and override *render_entity_title* method in `tuto/cubicweb_tuto/views.py`:

```
from cubicweb.predicates import is_instance
from cubicweb.web.views.primary import PrimaryView

class MuseumPrimaryView(PrimaryView):
    __select__ = is_instance("Museum")

    def render_entity_title(self, entity):
        """Renders the entity title.
        """
```

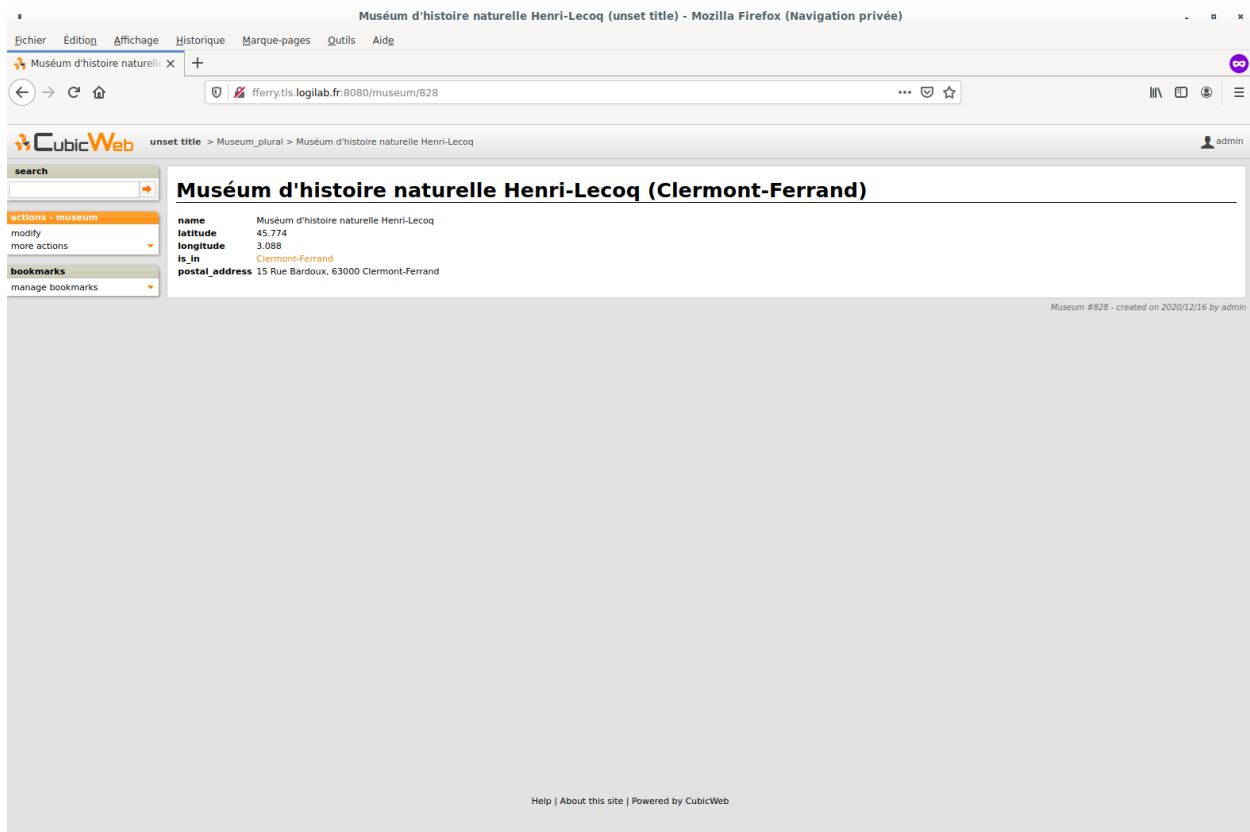
(continues on next page)

(continued from previous page)

```
city_name = entity.is_in[0].name
self.w(f"<h1>{entity.name} ({city_name})</h1>")
```

As stated before, CubicWeb comes with a system of views selection. This system is, among other things, based on selectors declared with `__select__` (you'll find more information about this in the [Registries and application objects](#) chapter). As we want to customize museum primary view, we use `__select__ = is_instance("Museum")` to tell CubicWeb this is only applicable when we display a *Museum* entity.

Then, we just override the method used to compute title to add the city name. To reach the city name, we use the relation *is_in* and choose the first and only one linked city, then ask for its name.



Use entities.py to add more logic

CubicWeb provides an ORM to easily programmatically manipulate entities. By default, entity types are instances of the `AnyEntity` class, which holds a set of predefined methods as well as property automatically generated for attributes/relations of the type it represents.

You can redefine each entity to provide additional methods or whatever you want to help you write your application. Customizing an entity requires that your entity:

- inherits from `cubicweb.entities.AnyEntity` or any subclass
- defines a `__regid__` linked to the corresponding data type of your schema

You may then want to add your own methods, override default implementation of some method, etc...

As we may want reuse our custom museum title (with city name, as defined in previous section), we will define it as a property of our *Museum* class.

To do so, write this code in `tuto/cubicweb_tuto/entities.py`:

```
from cubicweb.entities import AnyEntity, fetch_config

class Museum(AnyEntity):
    __regid__ = "Museum"

    @property
    def title_with_city(self):
        return f"{self.name} ({self.is_in[0].name})"
```

Then, we just have to use it our previously defined view in `tuto/cubicweb_tuto/views.py`:

```
from cubicweb.predicates import is_instance
from cubicweb.web.views.primary import PrimaryView

class MuseumPrimaryView(PrimaryView):
    __select__ = is_instance("Museum")

    def render_entity_title(self, entity):
        """Renders the entity title.
        """
        self.w(f"<h1>{entity.title_with_city}</h1>")
```

Conclusion

In this first part, we laid the cornerstone of our futur site, and discovered some core functionalities of *CubicWeb*. In next parts, we will improve views and see how to import all our data.

Enhance views

In *Getting started*, we saw how to develop our views by writing html code directly in CubicWeb views. In this part, we will see how to customize our web application using different methods : with pyramid views using jinja2 templates and with React.

Pyramid and Jinja2

React in a CubicWeb view

In this section, we want to add a map in museum pages to display where is the museum associated with the page.

To do this, we will use [React simple maps](#), a [React](#) library. Our goal is to add a react component inside our museum primary view.

First, we will setup our environment. At logilab, we use [Typescript](#) when it is possible, so we will use it also in this tutorial. As module builder, we will use [Webpack](#).

Thus, we need to create three files at the root of our cube: `package.json`, `tsconfig.json` and `webpack.config.js`. A lot of documentation can be find on the Web about how to configure a React/Typescript environment, so we are not going to dwell on it in this tutorial; and we will simply copy and paste the following files.

package.json:

```
{
  "name": "cubicweb_tuto",
  "version": "1.0.0",
  "description": "Summary ----- A cube for new CW tutorial",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack",
    "watch": "webpack --watch --mode=development"
  },
  "author": "Logilab",
  "license": "GPL-2.0-or-later",
  "dependencies": {
    "@types/react": "^17.0.0",
    "@types/react-dom": "^17.0.0",
    "@types/react-simple-maps": "^1.0.3",
    "prop-types": "^15.7.2",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-simple-maps": "^2.3.0",
    "ts-loader": "^8.0.14",
    "typescript": "^4.1.3",
    "webpack": "^5.18.0",
    "webpack-cli": "^4.4.0"
  }
}
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "jsx": "react",
    "strict": true,
    "esModuleInterop": true
  }
}
```

webpack.config.js:

```
const path = require("path");

module.exports = {
  entry: {
    "map.js": "./appjs/geomap.tsx",
  },
  output: {
    filename: "[name]",
    path: path.resolve(__dirname, "./cubicweb_tuto/data/")
  }
}
```

(continues on next page)

(continued from previous page)

```
},
resolve: {
  extensions: [".tsx", ".ts", ".jsx", ".js"]
},
module: {
  rules: [
    {
      test: [/\.tsx?$/],
      exclude: /node_modules/,
      use: ["ts-loader"]
    }
  ]
},
plugins: []
};
```

Now we have our configuration files, we have to install [NodeJS](#) and then install our project using *npm*.

```
sudo apt-get install nodejs
npm install
```

They are two last things to do:

- create a component to display a museum on the map;
- integrate our component in a CubicWeb view.

By convention, we put our js files in a *appjs* directory, and bundle are built in *cubicweb_tuto/data* (as you can see in our *webpack.config.js*). Then, we will create a file *geomap.tsx* in *appjs/*.

For our component, we will need three parameters: our museum name, its latitude and its longitude. These parameters will be defined in our CubicWeb view when we will call our script. Our file *geomap.tsx* can be written like this:

```
import React from 'react';
import ReactDOM from 'react-dom';
import {
  ComposableMap,
  Geographies,
  Geography,
  Marker,
  Point
} from "react-simple-maps";

const geoUrl = "https://raw.githubusercontent.com/zcreativelabs/react-simple-maps/master/
↳topojson-maps/world-110m.json";

declare const data: {
  name: string,
  latitude: number,
  longitude: number,
}

const MapChart = () => {
  return (
```

(continues on next page)

(continued from previous page)

```

<ComposableMap>
  <Geographies geography={geoUrl}>
    ({ geographies }) =>
      geographies
        .map(geo => (
          <Geography
            key={geo.rsmKey}
            geography={geo}
            fill="#EAEAE"
            stroke="#D6D6DA"
          />
        ))
  </Geographies>
  <Marker coordinates={[data.longitude, data.latitude] as Point}>
    <g
      fill="none"
      stroke="#FF5533"
      strokeWidth="2"
      strokeLinecap="round"
      strokeLinejoin="round"
      transform="translate(-12, -24)"
    >
      <circle cx="12" cy="10" r="3" />
      <path d="M12 21.7C17.3 17 20 13 20 10a8 8 0 1 0-16 0c0 3 2.7 6.9 8 11.7z" />
    </g>
    <text
      textAnchor="middle"
      y={10}
      style={{ fontFamily: "system-ui", fill: "#5D5A6D" }}
    >
      {data.name}
    </text>
  </Marker>
</ComposableMap>
);
};

function App() {
  return <MapChart/>
}

const root = document.getElementById("awesome-map");
ReactDOM.render(<App/>, root);

```

Now we will override the `render_entity(self, entity)` function of the Museum PrimaryView, in `cubicweb-tuto/views.py` to add:

- the bundle javascript including our component;
- a div with the id `awesome-map` which will be used by our component.

```
class MuseumPrimaryView(PrimaryView):
```

(continues on next page)

(continued from previous page)

```

__select__ = is_instance("Museum")

def render_entity(self, entity):
    self.render_entity_toolbox(entity)
    self.render_entity_title(entity)
    # entity's attributes and relations, excluding meta data
    # if the entity isn't meta itself
    if self.is_primary():
        boxes = self._prepare_side_boxes(entity)
    else:
        boxes = None
    if boxes or hasattr(self, "render_side_related"):
        self.w('<table width="100%"><tr><td style="width: 75%">')

        self.w('<div class="mainInfo">')
        self.content_navigation_components("navcontenttop")
        self.render_entity_attributes(entity)
        if self.main_related_section:
            self.render_entity_relations(entity)
        self.render_map(entity)
        self.content_navigation_components("navcontentbottom")
        self.w("</div>")
        # side boxes
        if boxes or hasattr(self, "render_side_related"):
            self.w("</td><td>")
            self.w('<div class="primaryRight">')
            self.render_side_boxes(boxes)
            self.w("</div>")
            self.w("</td></tr></table>")

    def render_entity_title(self, entity):
        """Renders the entity title, by default using entity's
        :meth:`dc_title()` method.
        """
        self.w(f"<h1>{entity.title_with_city}</h1>")

    def render_map(self, entity):
        """Renders a map displaying where the museum is."""
        if not (entity.latitude and entity.longitude):
            return
        js_file = f"{self._cw.vreg.config.datadir_url}map.js"
        data = json_dumps(entity)
        self.w('<div id="awesome-map"></div>')
        self.w(
            f"""
            <script type="text/javascript">
                const data = {data};
            </script>
            <script src={js_file}></script>
            """
        )

```

Most part of `render_entity(self, entity)` are the same as its definition in `PrimaryView`, except that we add a call to

`render_map(self, entity)`; which will add a *div* tag with a specific id and a *script* tag adding our javascript bundle, and define variables containing information to display a museum on the map. The specific id must be the same as the one we defined in our javascript file, *awesome-map*.

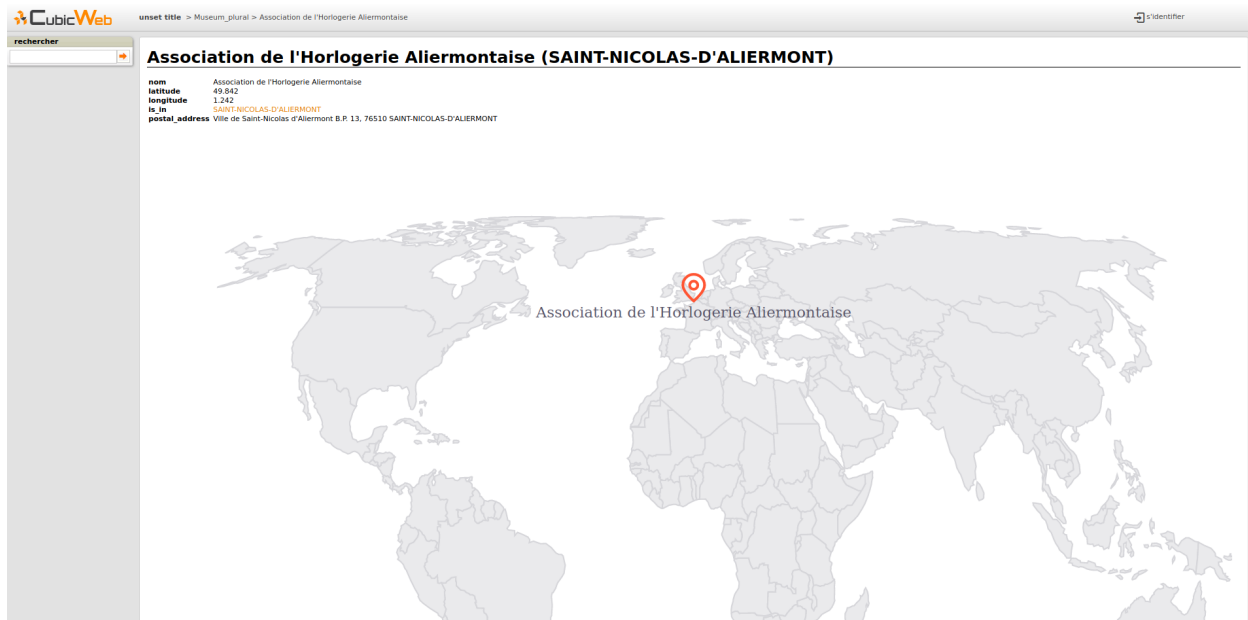
Now, it's time to build the javascript bundle using:

```
npm run build
```

And then, run our application:

```
cubicweb-ctl pyramid -D tuto_instance
```

We now have a world map displaying the location of our museum on museum pages. A lot of things could be done to have a better result, like center the map on the museum, but it's out of the scope of this tutorial.



React in a Pyramid view

Data management with CubicWeb

Import data

With our application customized, let's see how to import more data. There is several ways to import data in CubicWeb. In our tutorial, we want to import our museums from a *csv* file. This file is provided by the France's Ministry of Culture, and is available [here](#).

There are several ways to import data in CubicWeb; in this tutorial, we will use one of them, the others are described here: *Data Import*.

First of all, we define in `tuto/cubicweb_tuto/dataimport.py` a function which will read a file from a *filepath* and create the corresponding entities, using a *CubicWeb connection*:

```
import csv
```

(continues on next page)

(continued from previous page)

```

def import_museums(cnx, filepath):
    existing_cities = dict(cnx.execute("Any Z, C Where C is City, C zip_code Z"))
    existing_cities_nb = len(existing_cities)
    created_museum_nb = 0
    with open(filepath) as fileobj:
        reader = csv.DictReader(fileobj, delimiter=";")
        for record in reader:
            museum_name = record["NOM DU MUSEE"]
            street = record["ADR"]
            zip_code = record["CP"]
            city_name = record["VILLE"]
            try:
                lat, lng = record["coordonnees_finales"].split(",")
                lat_long = {
                    "latitude": lat,
                    "longitude": lng,
                }
            except (AttributeError, ValueError):
                lat_long = {}
            try:
                city = existing_cities[zip_code]
            except KeyError:
                city = cnx.create_entity("City", name=city_name, zip_code=zip_code)
                existing_cities[zip_code] = city.eid
            cnx.create_entity(
                "Museum",
                name=museum_name,
                postal_address=f"{street}, {zip_code} {city_name}",
                is_in=city,
                **lat_long,
            )
            created_museum_nb += 1

    print(
        "Import finished! {} existing cities, {} cities created, {} museums created.".
        ↪format(
            existing_cities_nb,
            len(existing_cities) - existing_cities_nb,
            created_museum_nb,
        )
    )

```

To be sure we don't have several time the same city, we first query CubicWeb to ask for all existing city. To do this, we use a specific language called **RQL**. As for SPARQL, it's a query language designed to query linked data. See [Introduction](#) for more information about it.

Then, we put existing cities in a dictionary, using zip code as key. In the real world, a zip code can concern several cities, but it's not really an issue in this tutorial.

For each line of our *csv* file, we will check if we already have the city in our base. If not, we create it. Then, we create our Museum entity with all needed arguments.

To create an entity, we use the *create_entity* method of the CubicWeb connection. This method takes as first argument the type of the entity (ie: the name of the corresponding class previously defined in *tuto/cubicweb_tuto/schema*).

py), and then all arguments of the entity type.

In our example, a city needs a name and a zip code. A museum needs a name, a postal address, a latitude, a longitude and a city. As *is_in* is a relation, we give to the corresponding argument the eid of the city.

Note: As we have defined Museum in the schema, we have to link each instance of Museum to a City, that's why we create the city before the museum, and give this city as argument of the museum.

If the city wasn't mandatory, we could add it later, using:

```
museum_entity.cw_set(is_in=city)
```

To use our function we need to create a CubicWeb command that will call it. First, we create a file `tuto/cubicweb_tuto/ccplugin.py` (the name doesn't matter, but it is commonly used for all new CubicWeb commands). Then, we write the following code:

```
from cubicweb.cwctl import CWCTL
from cubicweb.toolsutils import Command
from cubicweb.utils import adminctx

from cubicweb_tuto.dataimport import import_museums

@CWCTL.register
class ImportMuseums(Command):
    """
    Import Museums and Cities from a CSV from:
    https://data.culture.gouv.fr/explore/dataset/liste-et-localisation-des-musees-de-
    france/export/
    """

    arguments = "<instance> <csv_file>"
    name = "import-museums"
    min_args = max_args = 2

    def run(self, args):
        appid, csv_file = args[:2]

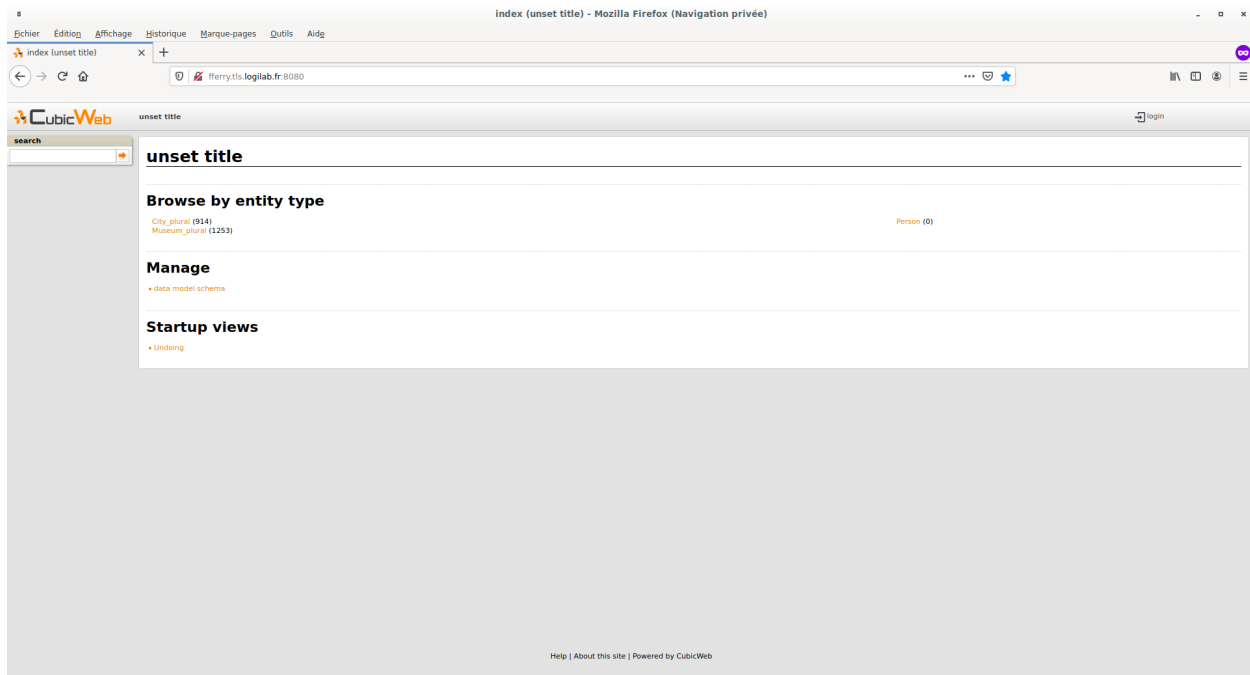
        with adminctx(appid) as cnx:
            import_museums(cnx, csv_file)
            cnx.commit()
```

- `@CWCTL.register` allows to register the command and then make it available with `cubicweb-ctl` command by its name.
- `arguments` defines which arguments take our command.
- `name` defines the name of the command.
- `with adminctx(appid) as cnx` allows to have an admin access to our instance, and then be able to create new entities.

Thus, to execute our import command, we just have to enter in our shell (within our virtual env):

```
cubicweb-ctl import-museums tuto_instance <path_to_the_csv>
```

After this script, we should be able to see that we have much more cities and museums by visiting the homepage of our CubicWeb instance:



RDF serialisation

Content negotiation

SETUP AND ADMINISTRATION

This part is for installation and administration of the *CubicWeb* framework and instances based on that framework.

4.1 Install a *CubicWeb* environment

Official releases are available from the [CubicWeb.org forge](#) and from [PyPI](#). Since CubicWeb is developed using [Agile software development](#) techniques, releases happen frequently. In a version numbered X.Y.Z, X changes after a few years when the API breaks, Y changes after a few weeks when features are added and Z changes after a few days when bugs are fixed.

There are several ways to install *CubicWeb* depending on your needs:

- *Using Docker*
- *In a virtualenv*
- *Using Pip*

If you are a power-user and need the very latest features, you can choose the following methods:

- *Downloading the source*
- *Using mercurial*

Additional configuration can be found in the section [Configure a CubicWeb environment](#) for better control and advanced features of *CubicWeb*.

4.1.1 Installing Dependencies

No matter your installation method, you will need to install the following Debian packages:

```
apt install gettext graphviz
```

`gettext` is used for translations (see [Internationalization](#)), and `graphviz` to display relation schemas within the web-site.

4.1.2 Installing *CubicWeb*

Docker install

Detailed instructions on how to deploy CubicWeb using docker can be found on the [docker hub](#).

Images are built using the source code available in the [docker-cubicweb](#) repository.

Virtualenv install

CubicWeb can be safely installed, used and contained inside a [virtualenv](#). To create and activate a [virtualenv](#), use the following commands:

```
pip install --user virtualenv
virtualenv venv
source venv/bin/activate
```

Then you can use either [pip](#) or `easy_install` to install *CubicWeb* inside an activated virtual environment.

pip install

[pip](#) is a python tool that helps downloading, building, installing, and managing Python packages and their dependencies. It is fully compatible with [virtualenv](#) and installs the packages from sources published on the [The Python Package Index](#).

A working compilation chain is needed to build modules which include C extensions. If you really do not want to compile anything, installing [lxml](#), and [libgecode](#) will help.

For Debian, these minimal dependencies can be obtained by doing:

```
apt install gcc python3-pip python3-dev python3-lxml
```

or, if you prefer to get as much as possible from pip:

```
apt install gcc python3-pip python3-dev libxslt1-dev libxml2-dev
```

For Windows, you can install pre-built packages (possible [source](#)). For a minimal setup, install:

- pip <http://www.lfd.uci.edu/~gohlke/pythonlibs/#pip>
- setuptools <http://www.lfd.uci.edu/~gohlke/pythonlibs/#setuptools>
- libxml-python <http://www.lfd.uci.edu/~gohlke/pythonlibs/#libxml-python>
- lxml <http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml>

Make sure to choose the correct architecture and version of Python.

Finally, install *CubicWeb* and its dependencies by running:

```
pip install cubicweb
```

Install from source

You can download the archive containing the sources from [CubicWeb forge downloads](#) section.

Make sure you also have all the *dependencies installed*.

Once uncompressed, you can install the framework from inside the uncompressed folder with:

```
python3 setup.py install
```

Or you can run *CubicWeb* directly from the source directory by setting the *resource mode* to *user*. This will ease the development with the framework.

Install from version control system

To keep-up with on-going development, clone the *Mercurial* repository:

```
hg clone -u 'last(tag())' https://forge.extranet.logilab.fr/cubicweb/cubicweb # stable ↵  
↵ version  
hg clone https://forge.extranet.logilab.fr/cubicweb/cubicweb # development branch
```

Make sure you also have all the *Installation dependencies*.

4.1.3 Installing *cubes*

Many other *cubes* are available. Those cubes can help expanding the functionalities offered by *CubicWeb*. A list is available at [PyPI](#) or at the [CubicWeb.org forge](#).

For example the *blog cube* can be installed using:

```
pip install cubicweb-blog
```

4.2 Configure a *CubicWeb* environment

You can *configure the database* system of your choice:

- *PostgreSQL configuration*
- *SQLite configuration*

For advanced features, have a look to:

- *Cubicweb resources configuration*

4.2.1 Databases configuration

Each instance can be configured with its own database connection information, that will be stored in the instance's sources file. The database to use will be chosen when creating the instance. CubicWeb is known to run with PostgreSQL (recommended) and SQLite.

Other possible sources of data include CubicWeb, LDAP and Mercurial, but at least one relational database is required for CubicWeb to work. You do not need to install a backend that you do not intend to use for one of your instances. SQLite is not fit for production use, but it works well for testing and ships with Python, which saves installation time when you want to get started quickly.

PostgreSQL

Many Linux distributions ship with the appropriate PostgreSQL packages. Basically, you need to install the following packages:

- *postgresql* and *postgresql-client*, which will pull the respective versioned packages (e.g. *postgresql-9.1* and *postgresql-client-9.1*) and, optionally,
- a *postgresql-plpython-X.Y* package with a version corresponding to that of the aforementioned packages (e.g. *postgresql-plpython-9.1*). (Not needed now by default)

If you run postgres on another host than the *CubicWeb* repository, you should install the *postgresql-client* package on the *CubicWeb* host, and others on the database host.

For extra details concerning installation, please refer to the [PostgreSQL project online documentation](#).

Database cluster

If you already have an existing cluster and PostgreSQL server running, you do not need to execute the initialization step of your PostgreSQL database unless you want a specific cluster for *CubicWeb* databases or if your existing cluster doesn't use the UTF8 encoding (see note below).

To initialize a PostgreSQL cluster, use the command `initdb`:

```
$ initdb -E UTF8 -D /path/to/pgsql
```

Note: `initdb` might not be in the PATH, so you may have to use its absolute path instead (usually something like `/usr/lib/postgresql/9.4/bin/initdb`).

Notice the encoding specification. This is necessary since *CubicWeb* usually want UTF8 encoded database. If you use a cluster with the wrong encoding, you'll get error like:

```
new encoding (UTF8) is incompatible with the encoding of the template database (SQL_
↳ ASCII)
HINT: Use the same encoding as in the template database, or use template0 as template.
```

Once initialized, start the database server PostgreSQL with the command:

```
$ postgres -D /path/to/pgsql
```

If you cannot execute this command due to permission issues, please make sure that your username has write access on the database.

```
$ chown username /path/to/pgsql
```


Database authentication

The database authentication is configured in *pg_hba.conf*. It can be either set to *ident sameuser* or *md5*. If set to *md5*, make sure to use an existing user of your database. If set to *ident sameuser*, make sure that your client's operating system user name has a matching user in the database. If not, please do as follow to create a user:

```
$ su
$ su - postgres
$ createuser -s -P <dbuser>
```

The option *-P* (for password prompt), will encrypt the password with the method set in the configuration file *pg_hba.conf*. If you do not use this option *-P*, then the default value will be null and you will need to set it with:

```
$ su postgres -c "echo ALTER USER <dbuser> WITH PASSWORD '<dbpassword>' | psql"
```

The above login/password will be requested when you will create an instance with *cubicweb-ctl create* to initialize the database of your instance.

Database creation

If you create the database by hand (instead of using the *cubicweb-ctl db-create* tool), you may want to make sure that the local settings are properly set. For example, if you need to handle french accents properly for indexing and sorting, you may need to create the database with something like:

```
$ createdb --encoding=UTF-8 --locale=fr_FR.UTF-8 -t template0 -O <owner> <dbname>
```

Notice that the *cubicweb-ctl db-create* does database initialization that may requires a postgres superuser. That's why a login/password is explicitly asked at this step, so you can use there a superuser without using this user when running the instance. Things that require special privileges at this step:

- database creation, require the 'create database' permission

Where *pgadmin* is a postgres superuser.

SQLite

SQLite has the great advantage of requiring almost no configuration. Simply use 'sqlite' as db-driver, and set path to the database as db-name. Don't specify anything for db-user and db-password, they will be ignore anyway.

Note: SQLite is great for testing and to play with cubicweb but is not suited for production environments.

4.2.2 Cubicweb resources configuration

Resource mode

Standard resource mode

A resource *mode* is a predefined set of settings for various resources directories, such as cubes, instances, etc. to ease development with the framework. There are two running modes with *CubicWeb*:

- **system:** resources are searched / created in the system directories (eg usually requiring root access):

- instances are stored in `<INSTALL_PREFIX>/etc/cubicweb.d`
 - temporary files (such as pid file) in `<INSTALL_PREFIX>/var/run/cubicweb`
- where `<INSTALL_PREFIX>` is the detected installation prefix (`'/usr/local'` for instance).

- **user:** resources are searched / created in the user home directory:
 - instances are stored in `~/etc/cubicweb.d`
 - temporary files (such as pid file) in `/tmp`

Within virtual environment

When installed within a virtualenv, CubicWeb will look for instances data as in user mode by default, that is in `$HOME/etc/cubicweb.d`. However the `CW_INSTANCES_DIR` environment variable should be preferably used.

Custom resource location

Notice that each resource path may be explicitly set using an environment variable if the default doesn't suit your needs. Here are the default resource directories that are affected according to mode:

- **system:**

```
CW_INSTANCES_DIR = <INSTALL_PREFIX>/etc/cubicweb.d/  
CW_INSTANCES_DATA_DIR = <INSTALL_PREFIX>/var/lib/cubicweb/instances/  
CW_RUNTIME_DIR = <INSTALL_PREFIX>/var/run/cubicweb/
```

- **user:**

```
CW_INSTANCES_DIR = ~/etc/cubicweb.d/  
CW_INSTANCES_DATA_DIR = ~/etc/cubicweb.d/  
CW_RUNTIME_DIR = /tmp
```

Cubes search path is also affected, see the [Cubes](#) section.

Setting Cubicweb Mode

By default, the mode is set to 'system' for standard installation. The mode is set to 'user' if *cubicweb is used from a mercurial repository*. You can force this by setting the `CW_MODE` environment variable to either 'user' or 'system' so you can easily:

- use system wide installation but user specific instances and all, without root privileges on the system (*export `CW_MODE=user`*)
- use local checkout of cubicweb on system wide instances (requires root privileges on the system (*export `CW_MODE=system`*))

If you've a doubt about the mode you're currently running, check the first line outputted by the `cubicweb-ctl list` command.

Development Mode (source)

If `.hg` directory is found into the cubicweb package, there are specific resource rules.

`<CW_SOFTWARE_ROOT>` is the source checkout's cubicweb directory:

- cubicweb migration files are searched in `<CW_SOFTWARE_ROOT>/misc/migration` instead of `<INSTALL_PREFIX>/share/cubicweb/migration/`.

Development Mode (virtualenv)

If a virtualenv is found to be activated (i.e. a `VIRTUAL_ENV` variable is found in environment), the virtualenv root is used as `<INSTALL_PREFIX>`. This, in particular, makes it possible to work in [setuptools development mode](#) (python `setup.py develop`) without any further configuration.

Environment configuration

Python

If you installed *CubicWeb* by cloning the Mercurial shell repository or from source distribution, then you will need to update the environment variable `PYTHONPATH` by adding the path to *cubicweb*:

Add the following lines to either `.bashrc` or `.bash_profile` to configure your development environment

```
export PYTHONPATH=/full/path/to/grshell-cubicweb
```

If you installed *CubicWeb* with packages, no configuration is required and your new cubes will be placed in `/usr/share/cubicweb/cubes` and your instances will be placed in `/etc/cubicweb.d`.

CubicWeb

Here are all environment variables that may be used to configure *CubicWeb*:

CW_MODE

Resource mode: user or system, as explained in [Resource mode](#).

CW_INSTANCES_DIR

Directory where cubicweb instances will be found.

CW_INSTANCES_DATA_DIR

Directory where cubicweb instances data will be written (backup file...)

CW_RUNTIME_DIR

Directory where pid files will be written

4.3 Deploy a *CubicWeb* application

4.3.1 Deployment with uwsgi

uWSGI is often used to deploy CubicWeb applications.

Short version is install *uwsgi*:

```
apt install uwsgi
```

Deploy a configuration file for your application */etc/uwsgi/apps-enabled/example.ini*. Don't forget to replace *example* with the instance name to deploy:

```
[uwsgi]
master = true
http = 0.0.0.0:8080
env = CW_INSTANCE=example
module = cubicweb.pyramid:wsgi_application()
processes = 2
threads = 8
plugins = http,python3
auto-procname = true
lazy-apps = true
log-master = true
disable-logging = true
```

You can run it manually with:

```
uwsgi --ini /etc/uwsgi/apps-enabled/example.ini
```

Apache configuration

It is possible to use apache (for example) as proxy in front of uwsgi.

For this to work you have to activate the following apache modules :

- rewrite
- proxy
- http_proxy

The command on Debian based systems for that is

```
a2enmod rewrite http_proxy proxy
/etc/init.d/apache2 restart
```

Example For an apache redirection of a site accessible via *http://localhost/demo* while cubicweb is actually running on port 8080::

```
ProxyPreserveHost On
RewriteEngine On
RewriteCond %{REQUEST_URI} ^/demo
RewriteRule ^/demo$ /demo/
RewriteRule ^/demo/(.*) http://127.0.0.1:8080/$1 [L,P]
```

and we will configure the *base-url* in the *all-in-one.conf* of the instance::

```
base-url = http://localhost/demo
```

4.3.2 Deployment with SaltStack

To deploy with SaltStack one can refer themselves to the [cubicweb-formula](#).

4.3.3 Deployment with Docker

To deploy in a docker container cluster, you should use our [docker image](#). The source code is also in the [forge](#). For a standard cube with no *apt* dependencies, the following dockerfile is fine:

```
FROM logilab/cubicweb:3.29
USER root
COPY . /src
RUN pip install -e /src
USER cubicweb
RUN docker-cubicweb-helper create-instance
```

To run your instance, don't forget the port redirection and change the image name:

```
docker run --rm -it -p 8080:8080 example:latest
```

If you need to customize the variables in the files *all-in-one.conf* or *sources*, you should pass them as environment variables. For example, the database name is read from *CW_DB_NAME*. The admin password is read from *CW_PASSWORD*. Also if the database is on the host, it has to be accessible from the container:

```
docker run --rm -it -p 8080:8080 --env-file ./env -v /var/run/postgresql:/var/run/
↳postgresql example:latest
```

If your instance needs a scheduler, it has to be run in a separate container from the same image:

```
docker run --rm -it --env-file ./env -v /var/run/postgresql:/var/run/postgresql
↳example:latest cubicweb-ctl scheduler instance
```

Don't forget to change the image name *example:latest* and the instance name *name*.

4.3.4 Deployment with Kubernetes

To deploy in a Kubernetes cluster, you can take inspiration from the [deploy instructions](#) included in the [fresh cube](#). It includes nginx to serve static files, one container for the application and one for the scheduler and also an [initContainer](#) to automatically upgrade the database in case of new version.

4.4 cubicweb-ctl tool

cubicweb-ctl is the swiss knife to manage *CubicWeb* instances. The general syntax is

```
cubicweb-ctl <command> [options command] <arguments commands>
```

To view available commands

```
cubicweb-ctl
cubicweb-ctl --help
```

Please note that the commands available depends on the *CubicWeb* packages and cubes that have been installed.

To view the help menu on specific command

```
cubicweb-ctl <command> --help
```

4.4.1 Listing available cubes and instance

- `list`, provides a list of the available configuration, cubes and instances.

4.4.2 Creation of a new cube

Create your new cube cube

```
cubicweb-ctl newcube -d <target directory>
```

This will create a new cube <target directory>.

4.4.3 Create an instance

You must ensure `~/etc/cubicweb.d/` exists prior to this. On windows, the `'~'` part will probably expand to 'Documents and Settings/user'.

To create an instance from an existing cube, execute the following command

```
cubicweb-ctl create <cube_name> <instance_name>
```

This command will create the configuration files of an instance in `~/etc/cubicweb.d/<instance_name>`.

The tool `cubicweb-ctl` executes the command `db-create` and `db-init` when you run `create` so that you can complete an instance creation in a single command. But of course it is possible to issue these separate commands separately, at a later stage.

4.4.4 Command to create/initialize an instance database

- `db-create`, creates the system database of an instance (tables and extensions only)
- `db-init`, initializes the system database of an instance (schema, groups, users, workflows...)

4.4.5 Run an instance

To start an instance during development, use

```
cubicweb-ctl pyramid [-D] [-l <log-level>] <instance-id>
```

without `-D`, the instance will be start in the background, as a daemon.

See *The 'pyramid' command* for more details.

In production, it is recommended to run CubicWeb through a WSGI server like uWSGI or Gunicorn. See *cubicweb.pyramid* more details.

4.4.6 Commands to maintain instances

- `upgrade`, launches the existing instances migration when a new version of *CubicWeb* or the cubes installed is available
- `shell`, opens a (Python based) migration shell for manual maintenance of the instance
- `db-dump`, creates a dump of the system database
- `db-restore`, restores a dump of the system database
- `db-check`, checks data integrity of an instance. If the automatic correction is activated, it is recommended to create a dump before this operation.
- `schema-sync`, synchronizes the persistent schema of an instance with the instance schema. It is recommended to create a dump before this operation.

4.4.7 Commands to maintain i18n catalogs

- `i18ncubicweb`, regenerates messages catalogs of the *CubicWeb* library
- `i18ncube`, regenerates the messages catalogs of a cube
- `i18ninstance`, recompiles the messages catalogs of an instance. This is automatically done while upgrading.

See also chapter *Internationalization*.

4.4.8 Other commands

- `delete`, deletes an instance (configuration files and database)

4.5 Creation of your first instance

4.5.1 Instance creation

Now that we created a cube, we can create an instance and access it via a web browser. We will use a *all-in-one* configuration to simplify things

```
cubicweb-ctl create -c all-in-one mycube myinstance
```

Note: Please note that we created a new cube for a demo purposes but you could have used an existing cube available in our standard library such as *blog* or *person* for example.

A series of questions will be prompted to you, the default answer is usually sufficient. You can anyway modify the configuration later on by editing configuration files. When a login/password are requested to access the database please use the credentials you created at the time you configured the database (*PostgreSQL*).

It is important to distinguish here the user used to access the database and the user used to login to the cubicweb instance. When an instance starts, it uses the login/password for the database to get the schema and handle low level transaction. But, when **cubicweb-ctl create** asks for a manager login/psswd of *CubicWeb*, it refers to the user you will use during the development to administrate your web instance. It will be possible, later on, to use this user to create other users for your final web instance.

4.5.2 Instance administration

start / stop

When this command is completed, the definition of your instance is located in `~/etc/cubicweb.d/myinstance/*`. To launch it, you just type

```
cubicweb-ctl pyramid -D myinstance
```

The option `-D` specifies the *debug mode* : the instance is not running in server mode and does not disconnect from the terminal, which simplifies debugging in case the instance is not properly launched. You can see how it looks by visiting the URL `http://localhost:8080` (the port number depends of your configuration). To login, please use the cubicweb administrator login/password you defined when you created the instance.

To shutdown the instance, Ctrl-C in the terminal window is enough. If you did not use the option `-D`, then type

```
cubicweb-ctl stop myinstance
```

This is it! All is settled down to start developping your data model...

Note: The output of `cubicweb-ctl pyramid -D myinstance` can be overwhelming. It is possible to reduce the log level with the `-loglevel` parameter as in `cubicweb-ctl pyramid -D myinstance -l info` to filter out all logs under *info* gravity.

upgrade

A manual upgrade step is necessary whenever a new version of CubicWeb or a cube is installed, in order to synchronise the instance's configuration and schema with the new code. The command is:

```
cubicweb-ctl upgrade myinstance
```

A series of questions will be asked. It always starts with a proposal to make a backup of your sources (where it applies). Unless you know exactly what you are doing (i.e. typically fiddling in debug mode, but definitely NOT migrating a production instance), you should answer YES to that.

The remaining questions concern the migration steps of *CubicWeb*, then of the cubes that form the whole application, in reverse dependency order.

In principle, if the migration scripts have been properly written and tested, you should answer YES to all questions.

Sometimes, typically while debugging a migration script, something goes wrong and the migration fails. Unfortunately the database may be in an incoherent state. You have two options here:

- fix the bug, restore the database and restart the migration process from scratch (quite recommended in a production environment)
- try to replay the migration up to the last successful commit, that is answering NO to all questions up to the step that failed, and finish by answering YES to the remaining questions.

4.6 Configure an instance

While creating an instance, a configuration file is generated in:

```
$ (CW_INSTANCES_DIR) / <instance> / <configuration name>.conf
```

For example:

```
/etc/cubicweb.d/myblog/all-in-one.conf
```

It is a simple text file in the INI format (http://en.wikipedia.org/wiki/INI_file). In the following description, each option name is prefixed with its own section and followed by its default value if necessary, e.g. “<section>.<option> [value].”

Note: At runtime, configuration options can be overridden by environments variables which name follows the option name with - replaced by _ and a CW_ prefix. For instance CW_BASE_URL=https://www.example.com would override the base-url configuration option.

4.6.1 Configuring the Web server

web.auth-model [cookie] authentication mode, cookie or http

web.realm realm of the instance in http authentication mode

web.http-session-time [0] period of inactivity of an HTTP session before it closes automatically. Duration in seconds, 0 meaning no expiration (or more exactly at the closing of the browser client)

main.anonymous-user, main.anonymous-password login and password to use to connect to the RQL server with HTTP anonymous connection. CWUser account should exist.

main.base-url url base site to be used to generate the urls of web pages

4.6.2 Setting up the web client

web.embed-allowed regular expression matching sites which could be “embedded” in the site (controllers ‘embed’)

web.submit-url url where the bugs encountered in the instance can be mailed to

4.6.3 RQL server configuration

main.host host name if it can not be detected correctly

main.pid-file file where will be written the server pid

main.uid user account to use for launching the server when it is root launched by init

main.session-time [30*60] timeout of a RQL session

main.query-log-file file where all requests RQL executed by the server are written

4.6.4 Configuring e-mail

RQL and web server side:

email.mangle-mails [no] indicates whether the email addresses must be displayed as is or transformed

RQL server side:

email.smtp-host [mail] hostname hosting the SMTP server to use for outgoing mail

email.smtp-port [25] SMTP server port to use for outgoing mail

email.smtp-username SMTP server username for authenticated email sending

email.smtp-password SMTP server password for authenticated email sending

email.sender-name name to use for outgoing mail of the instance

email.sender-addr address for outgoing mail of the instance

email.default dest-addrs destination addresses by default, if used by the configuration of the dissemination of the model (separated by commas)

email.supervising-addrs destination addresses of e-mails of supervision (separated by commas)

4.6.5 Configuring logging

main.log-threshold level of filtering messages (DEBUG, INFO, WARNING, ERROR)

main.log-file file to write messages

4.6.6 Configuring persistent properties

Other configuration settings are in the form of entities *CWProperty* in the database. It must be edited via the web interface or by RQL queries.

ui.encoding Character encoding to use for the web

navigation.short-line-size number of characters for “short” display

navigation.page-size maximum number of entities to show per results page

navigation.related-limit number of related entities to show up on primary entity view

navigation.combobox-limit number of entities unrelated to show up on the drop-down lists of the sight on an editing entity view

4.6.7 Cross-Origin Resource Sharing

CubicWeb’s support for the [CORS](#) protocol is provided by the [wsgicors](#) middleware at the Pyramid level. For now, the provided implementation only deals with access to a CubicWeb instance as a whole. Support for a finer granularity may be considered in the future.

A few parameters can be set to configure the [CORS](#) capabilities of CubicWeb, the values are passed to the *wsgi-cors.CORS()* middleware constructor, hence the [wsgicors](#) documentation can be used for more details.

access-control-allow-origin comma-separated list of allowed origin domains or “*” for any domain

access-control-allow-methods comma-separated list of allowed HTTP methods

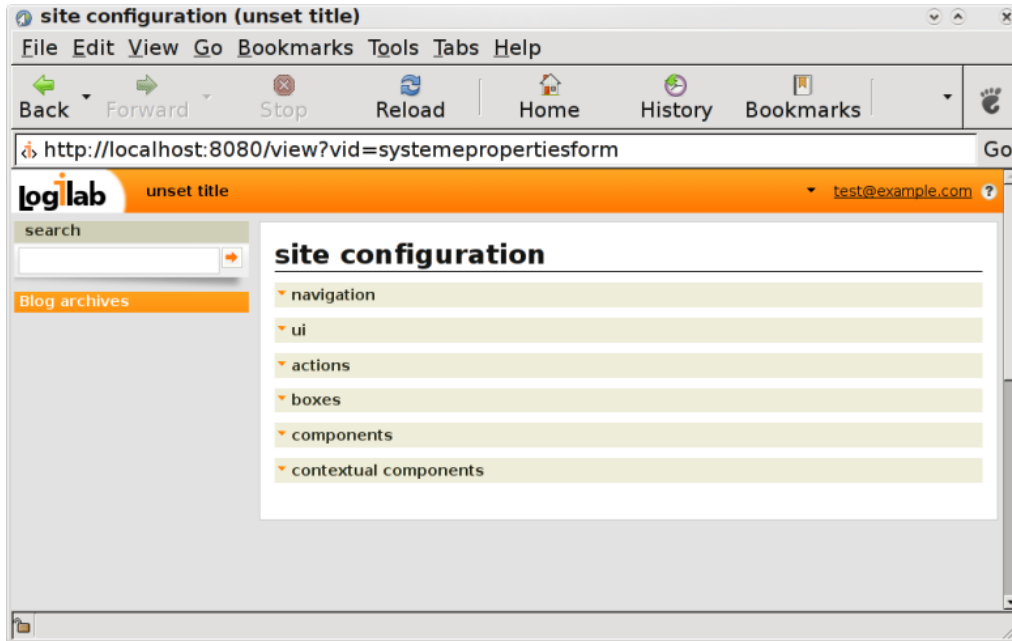
access-control-allow-headers comma-separated list of allowed HTTP custom headers (used in simple requests)

access-control-expose-headers comma-separated list of allowed HTTP custom headers (used in preflight requests)

access-control-max-age maximum age of cross-origin resource sharing (in seconds)

credentials is always set to *true* and is not configurable.

4.7 User interface for web site configuration



This panel allows you to configure the appearance of your instance site. Six menus are available and we will go through each of them to explain how to use them.

4.7.1 Navigation

This menu provides you a way to adjust some navigation options depending on your needs, such as the number of entities to display by page of results. Follows the detailed list of available options :

- `navigation.combobox-limit` : maximum number of entities to display in related combo box (sample format: 23)
- `navigation.page-size` : maximum number of objects displayed by page of results (sample format: 23)
- `navigation.related-limit` : maximum number of related entities to display in the primary view (sample format: 23)
- `navigation.short-line-size` : maximum number of characters in short description (sample format: 23)

4.7.2 UI

This menu provides you a way to customize the user interface settings such as date format or encoding in the produced html. Follows the detailed list of available options :

- `ui.date-format` : how to format date in the ui (“man strftime” for format description)
- `ui.datetime-format` : how to format date and time in the ui (“man strftime” for format description)
- `ui.default-text-format` : default text format for rich text fields.
- `ui.encoding` : user interface encoding
- `ui.fckeditor` : should html fields being edited using fckeditor (a HTML WYSIWYG editor). You should also select text/html as default text format to actually get fckeditor.
- `ui.float-format` : how to format float numbers in the ui

- `ui.language` : language of the user interface
- `ui.main-template` : id of main template used to render pages
- `ui.site-title` : site title, which is displayed right next to the logo in the header
- `ui.time-format` : how to format time in the ui (“man strftime” for format description)

4.7.3 Actions

This menu provides a way to configure the context in which you expect the actions to be displayed to the user and if you want the action to be visible or not. You must have notice that when you view a list of entities, an action box is available on the left column which display some actions as well as a drop-down menu for more actions.

The context available are :

- `mainactions` : actions listed in the left box
- `moreactions` : actions listed in the *more* menu of the left box
- `addrelated` : add actions listed in the left box
- `useractions` : actions listed in the first section of drop-down menu accessible from the right corner user login link
- `siteactions` : actions listed in the second section of drop-down menu accessible from the right corner user login link
- `hidden` : select this to hide the specific action

4.7.4 Boxes

The instance has already a pre-defined set of boxes you can use right away. This configuration section allows you to place those boxes where you want in the instance interface to customize it.

The available boxes are :

- `actions box` : box listing the applicable actions on the displayed data
- `boxes_blog_archives_box` : box listing the blog archives
- `possible views box` : box listing the possible views for the displayed data
- `rss box` : RSS icon to get displayed data as a RSS thread
- `search box` : search box
- `startup views box` : box listing the configuration options available for the instance site, such as *Preferences* and *Site Configuration*

4.7.5 Components

[WRITE ME]

4.7.6 Contextual components

[WRITE ME]

4.8 Multiple sources of data

Data sources include SQL, LDAP, RQL, mercurial and subversion.

4.9 LDAP integration

4.9.1 Overview

Using LDAP as a source for user credentials and information is quite easy. The most difficult part lies in building an LDAP schema or using an existing one.

At cube creation time, one is asked if more sources are wanted. LDAP is one possible option at this time. Of course, it is always possible to set it up later using the *CWSource* entity type, which we discuss there.

It is possible to add as many LDAP sources as wanted, which translates in as many *CWSource* entities as needed.

The general principle of the LDAP source is, given a proper configuration, to create local users matching the users available in the directory and deriving local user attributes from directory users attributes. Then a periodic task ensures local user information synchronization with the directory.

Users handled by such a source should not be edited directly from within the application instance itself. Rather, updates should happen at the LDAP server level.

Credential checks are `_always_` done against the LDAP server.

Note: There are currently two ldap source types: the older *ldapuser* and the newer *ldapfeed*. The older will be deprecated anytime soon, as the newer has now gained all the features of the old and does not suffer from some of its illnesses.

The *ldapfeed* creates real *CWUser* entities, and then activate/deactivate them depending on their presence/absence in the corresponding LDAP source. Their attribute and state (activated/deactivated) are hence managed by the source mechanism; they should not be altered by other means (as such alterations may be overridden in some subsequent source synchronisation).

4.9.2 Configuration of an LDAPfeed source

Additional sources are created at cube creation time or later through the user interface.

Configure an *ldapfeed* source from the user interface under *Manage* then *data sources*:

- At this point *type* has been set to *ldapfeed*.
- The *parser* attribute shall be set to *ldapfeed*.
- The *url* attribute shall be set to an URL such as `ldap://ldapsrvr.domain/`.
- The *configuration* attribute contains many options. They are described in detail in the next paragraph.

4.9.3 Options of an LDAPfeed source

Let us enumerate the options by categories (LDAP server connection, LDAP schema mapping information).

LDAP server connection options:

- *auth-mode*, (choices are simple, cram_md5, digest_md5, gssapi, support for the later being partial as of now)
- *auth-realm*, realm to use when using gssapi/kerberos authentication
- *data-cnx-dn*, user dn to use to open data connection to the ldap (eg used to respond to rql queries)
- *data-cnx-password*, password to use to open data connection to the ldap (eg used to respond to rql queries)
- *start-tls*, starting TLS before bind (valid values: “true”, “false”)

If the LDAP server accepts anonymous binds, then it is possible to leave data-cnx-dn and data-cnx-password empty. This is, however, quite unlikely in practice. Beware that the LDAP server might hide attributes such as “userPassword” while the rest of the attributes remain visible through an anonymous binding.

LDAP schema mapping options:

- *user-base-dn*, base DN to lookup for users
- *user-scope*, user search scope (valid values: “BASE”, “ONELEVEL”, “SUBTREE”)
- *user-classes*, classes of user (with Active Directory, you want to say “user” here)
- *user-filter*, additional filters to be set in the ldap query to find valid users
- *user-login-attr*, attribute used as login on authentication (with Active Directory, you want to use “sAMAccountName” here)
- *user-default-group*, name of a group in which ldap users will be by default. You can set multiple groups by separating them by a comma
- *user-attrs-map*, map from ldap user attributes to cubicweb attributes (with Active Directory, you want to use sAMAccountName:login,mail:email,givenName:firstname,sn:surname)

4.9.4 Other notes

- Cubicweb is able to start if ldap cannot be reached. . . If some source ldap server cannot be used while an instance is running, the corresponding users won’t be authenticated but their status will not change (e.g. they will not be deactivated)
- The user-base-dn is a key that helps cubicweb map CWUsers to LDAP users: beware updating it
- When a user is removed from an LDAP source, it is deactivated in the CubicWeb instance; when a deactivated user comes back in the LDAP source, it (automatically) is activated again
- You can use the CWSourceHostConfig to have variants for a source configuration according to the host the instance is running on. To do so go on the source’s view from the sources management view.

4.10 RQL logs

You can configure the *CubicWeb* instance to keep a log of the queries executed against your database. To do so, edit the configuration file of your instance `.../etc/cubicweb.d/myapp/all-in-one.conf` and uncomment the variable `query-log-file`:

```
# web instance query log file
query-log-file=/tmp/rql-myapp.log
```


BACKEND DEVELOPMENT

This part is about developing applications with the *CubicWeb* framework. It is not concerned with the web system, which is a separate layer and has its own whole chapter.

5.1 Cubes

This chapter describes how to define your own cubes and reuse already available cubes.

5.1.1 What is a Cube?

A cube is the equivalent of a module or a component but for CubicWeb. A website made with CubicWeb is generally an instance based on a cube that is just an assembly of already existing cubes with some domain specific logics. If you need a functionality that is not available, then you write a new Cube for it (and hopefully share it with the community).

A cube is made of mainly four parts:

- its *data model* stored in *schema.py*,
- its logic added to the data stored in *entities*
- its logic concerning dataflow stored in *hooks*,
- its user interface stored in *views*.

A cube can also define:

- its internationalization stored in the *i18n/*,
- new cubicweb commands stored in *ccplugin.py*.

A cube can use other cubes as building blocks and assemble them to provide a whole with richer fonctionnalités than its parts. The cubes `cubicweb-blog` and `cubicweb-comment` could be used to make a cube named *myblog* with commentable blog entries.

The [CubicWeb.org Forge](#) offers a large number of cubes developed by the community and available under a free software license. They are designed with the [KISS principle](#) as each cube usually adds a single functionality. Usually an application is an instance based on a regular cube that is just an assembly of existing cubes with some specific logics.

Note: The command `cubicweb-ctl list` displays the list of available cubes.

5.1.2 Standard structure for a cube

A cube named “mycube” is a Python package “cubicweb-mycube” structured as follows:

```
cubicweb-mycube/
├── cubicweb_mycube
│   ├── data
│   ├── entities.py
│   ├── hooks.py
│   ├── i18n
│   │   ├── de.po
│   │   ├── en.po
│   │   ├── es.po
│   │   └── fr.po
│   ├── __init__.py
│   ├── migration
│   │   └── postcreate.py
│   ├── __pkginfo__.py
│   ├── schema.py
│   └── views.py
├── cubicweb-mycube.spec
├── MANIFEST.in
├── README.rst
├── setup.py
├── test
│   ├── data
│   │   └── bootstrap_cubes
│   ├── __pycache__
│   └── test_mycube.py
└── tox.ini
```

We can use subpackages instead of Python modules for `views.py`, `entities.py`, `schema.py` or `hooks.py`. For example, we could have:

```
cubicweb-mycube/
|
|-- cubicweb_mycube/
|   |
|   |-- entities.py
|   |-- hooks.py
|   `-- views/
|       |-- __init__.py
|       |-- forms.py
|       |-- primary.py
|       `-- widgets.py
```

where :

- `schema` contains the schema definition (server side only)
- `entities` contains the entity definitions (server side and web interface)
- `hooks` contains hooks and/or views notifications (server side only)
- `views` contains the web interface components (web interface only)
- `test` contains tests related to the cube

- `i18n` contains message catalogs for supported languages (server side and web interface)
- `data` contains data files for static content (images, css, javascript code)... (web interface only)
- `migration` contains initialization files for new instances (`postcreate.py`) and a file containing dependencies of the component depending on the version (`depends.map`)
- file `__pkginfo__.py` provides component meta-data, especially the distribution and the current version (server side and web interface) or sub-cubes used by the cube.

At least you should have the file `__pkginfo__.py`.

The `site_cubicweb.py` files

It contains the definition of cube options that are customisable through *all-in-one.conf*. This should define the attribute *options* :

```
# -*- coding: utf-8 -*-
# cubicweb-mycube/cubicweb_mycube/site_cubicweb.py

options = (
    ('example-option-name-1',
     {'type': 'string',
      'default': 'Default value',
      'help': 'Some text explaining the usage of this option.',
      'group': 'cubicweb_mycube',
      'level': 2,
     }),
)
```

The options format are defined in *logilab common*. The *options* attribute should be a list of ('option-name', option-value). The value should be dict with the following entries:

- **type**: available types are : string, int, float, file, font, color, regexp, csv, yn (yes/no), bool, named, password, date, time, bytes, choice and multiple_choice.
- **default**: the default value of the option.
- **help**: the message to print as a help message.
- **group**: the section where the option should be stored in the `all-in-one.conf`.
- **level**: the verbosity at which the help should be displayed.

This is useful to add token configuration or endpoint, see for example *sentry* or *seo*.

When modifying this, don't forget to add a *migration script*.

The `__pkginfo__.py` file

It contains metadata describing your cube, mostly useful for packaging.

Two important attributes of this module are `__depends__` and `__recommends__` dictionaries that indicates what should be installed (and each version if necessary) for the cube to work.

Dependency on other cubes are expected to be of the form `'cubicweb-<cubeName>'`.

When an instance is created, dependencies are automatically installed, while recommends are not.

Recommends may be seen as a kind of 'weak dependency'. Eg, the most important effect of recommending a cube is that, if cube A recommends cube B, the cube B will be loaded before the cube A (same thing happen when A depends on B).

Having this behaviour is sometime desired: on schema creation, you may rely on something defined in the other's schema; on database creation, on something created by the other's postcreate, and so on.

5.1.3 The `setup.py` file

This is standard `setuptools` based setup module which reads most of its data from `__pkginfo__.py`. In the `setup` function call, it should also include an entry point definition under the `cubicweb.cubes` group so that CubicWeb can discover cubes (in particular their custom `cubicweb-ctl` commands):

```
setup(
    # ...
    entry_points={
        'cubicweb.cubes': [
            'mycube=cubicweb_mycube',
        ],
    },
    # ...
)
```

The `__init__.py` file

The first purpose of this file is to define the cube as a python module.

Furthermore, this file is, by default, the starting point of pyramid mechanism of inclusion for routes, views, predicates, etc. During initialization, Pyramid will check for the `includeme` function in this file. See [the documentation of pyramid](#).

`migration/precreate.py` and `migration/postcreate.py`

The precreate script is executed at instance creation time or when the cube is added to an existing instance, before the schema is serialized. This is typically to create groups referenced by the cube's schema.

The postcreate script, executed at instance creation time or when the cube is added to an existing instance. You could setup site properties or a workflow here for example.

More information : [:doc:`see migration description <book/devrepo/migration.rst>`_](#)

External resources such as image, javascript and css files

Out-of the box testing

Packaging and distribution

5.1.4 Creating a new cube from scratch

Let's start by creating the cube environment in which we will develop

```
cd ~/myproject
# use cubicweb-ctl to generate a template for the cube
# will ask some questions, most with nice default
cubicweb-ctl newcube mycube
# makes the cube source code managed by mercurial
cd cubicweb-mycube
hg init
hg add .
hg ci
```

If all went well, you should see the cube you just created in the list returned by `cubicweb-ctl list` in the *Available cubes* section. If not, please refer to *Environment configuration*.

To reuse an existing cube, add it to the list named `__depends_cubes__` which is defined in `__pkginfo__.py`. This variable is used for the instance packaging (dependencies handled by system utility tools such as APT) and to find used cubes when the database for the instance is created.

5.1.5 Available cubes

An instance is made of several basic cubes. In the set of available basic cubes we can find for example:

Base entity types

- **addressbook**: PhoneNumber and PostalAddress
- **card**: Card, generic documenting card
- **event**: Event (define events, display them in calendars)
- **file**: File (to allow users to upload and store binary or text files)
- **link**: Link (to collect links to web resources)
- **mailinglist**: MailingList (to reference a mailing-list and the URLs for its archives and its admin interface)
- **person**: Person (easily mixed with addressbook)
- **task**: Task (something to be done between start and stop date)
- **zone**: Zone (to define places within larger places, for example a city in a state in a country)

Classification

- **folder**: Folder (to organize things by grouping them in folders)
- **keyword**: Keyword (to define classification schemes)
- **tag**: Tag (to tag anything)

Other features

- **basket**: Basket (like a shopping cart)
- **blog**: a blogging system using Blog and BlogEntry entity types
- **comment**: system to attach comment threads to entities)
- **email**: archiving management for emails (*Email*, *Emailpart*, *Emailthread*), trigger action in cubicweb through email

To declare the use of a cube, once installed, add the name of the cube and its dependency relation in the `__depends_cubes__` dictionary defined in the file `__pkginfo__.py` of your own component.

5.2 The Registry, selectors and application objects

This chapter deals with some of the core concepts of the *CubicWeb* framework which make it different from other frameworks (and maybe not easy to grasp at a first glance). To be able to do advanced development with *CubicWeb* you need a good understanding of what is explained below.

This chapter goes deep into details. You don't have to remember them all but keep it in mind so you can go back there later.

An overview of AppObjects, the VRegistry and Selectors is given in the *Registries and application objects* chapter.

5.2.1 The CWRegistryStore

The *CWRegistryStore* can be seen as a two-level dictionary. It contains all dynamically loaded objects (subclasses of *AppObject*) to build a *CubicWeb* application. Basically:

- the first level key returns a *registry*. This key corresponds to the `__registry__` attribute of application object classes
- the second level key returns a list of application objects which share the same identifier. This key corresponds to the `__regid__` attribute of application object classes.

A *registry* holds a specific kind of application objects. There is for instance a registry for entity classes, another for views, etc...

The *CWRegistryStore* has two main responsibilities:

- being the access point to all registries
- handling the registration process at startup time, and during automatic reloading in debug mode.

Details of the recording process

On startup, *CubicWeb* loads application objects defined in its library and in cubes used by the instance. Application objects from the library are loaded first, then those provided by cubes are loaded in dependency order (e.g. if your cube depends on an other, objects from the dependency will be loaded first). The layout of the modules or packages in a cube is explained in *Standard structure for a cube*.

For each module:

- by default all objects are registered automatically
- if some objects have to replace other objects, or have to be included only if some condition is met, you'll have to define a *registration_callback(vreg)* function in your module and explicitly register **all objects** in this module, using the api defined below.

Note: Once the function *registration_callback(vreg)* is implemented in a module, all the objects from this module have to be explicitly registered as it disables the automatic objects registration.

API for objects registration

Here are the registration methods that you can use in the *registration_callback* to register your objects to the *CWRegistryStore* instance given as argument (usually named *vreg*):

- *register_all()*
- *register_and_replace()*
- *register()*
- *unregister()*

Examples:

```
# web/views/basecomponents.py
def registration_callback(vreg):
    # register everything in the module except SeeAlsoComponent
    vreg.register_all(globals().itervalues(), __name__, (SeeAlsoVComponent,))
    # conditionally register SeeAlsoVComponent
    if 'see_also' in vreg.schema:
        vreg.register(SeeAlsoVComponent)
```

In this example, we register all application object classes defined in the module except *SeeAlsoVComponent*. This class is then registered only if the 'see_also' relation type is defined in the instance's schema.

```
# goa/appobjects/sessions.py
def registration_callback(vreg):
    vreg.register(SessionsCleaner)
    # replace AuthenticationManager by GAEAuthenticationManager
    vreg.register_and_replace(GAEAuthenticationManager, AuthenticationManager)
    # replace PersistentSessionManager by GAEPersistentSessionManager
    vreg.register_and_replace(GAEPersistentSessionManager, PersistentSessionManager)
```

In this example, we explicitly register classes one by one:

- the *SessionCleaner* class
- the *GAEAuthenticationManager* to replace the *AuthenticationManager*

- the *GAPersistentSessionManager* to replace the *PersistentSessionManager*

If at some point we register a new appobject class in this module, it won't be registered at all without modification to the *registration_callback* implementation. The previous example will register it though, thanks to the call to the *register_all* method.

Runtime objects selection

Now that we have all application objects loaded, the question is : when I want some specific object, for instance the primary view for a given entity, how do I get the proper object ? This is what we call the **selection mechanism**.

As explained in the *The Core Concepts of CubicWeb* section:

- each application object has a **selector**, defined by its `__select__` class attribute
- this selector is responsible to return a **score** for a given context
 - 0 score means the object doesn't apply to this context
 - else, the higher the score, the better the object suits the context
- the object with the highest score is selected.

Note: When no single object has the highest score, an exception is raised in development mode to let you know that the engine was not able to identify the view to apply. This error is silenced in production mode and one of the objects with the highest score is picked.

In such cases you would need to review your design and make sure your selectors or appobjects are properly defined. Such an error is typically caused by either forgetting to change the `__regid__` in a derived class, or by having copy-pasted some code.

For instance, if you are selecting the primary (`__regid__ = 'primary'`) view (`__registry__ = 'views'`) for a result set containing a *Card* entity, two objects will probably be selectable:

- the default primary view (`__select__ = is_instance('Any')`), meaning that the object is selectable for any kind of entity type
- the specific *Card* primary view (`__select__ = is_instance('Card')`), meaning that the object is selectable for *Card* entities

Other primary views specific to other entity types won't be selectable in this case. Among selectable objects, the `is_instance('Card')` selector will return a higher score since it's more specific, so the correct view will be selected as expected.

API for objects selections

Here is the selection API you'll get on every registry. Some of them, as the 'etypes' registry, containing entity classes, extend it. In those methods, **args*, ***kwargs* is what we call the **context**. Those arguments are given to selectors that will inspect their content and return a score accordingly.

`select()`

`select_or_none()`

`possible_objects()`

`object_by_id()`

5.2.2 The *AppObject* class

The `cubicweb.appobject.AppObject` class is the base class for all dynamically loaded objects (application objects) accessible through the `cubicweb.cwvreg.CWRegistryStore`.

5.2.3 Predicates and selectors

Predicates are scoring functions that are called by the registry to tell whenever an appobject can be selected in a given context. Predicates may be chained together using operators to build a selector. A selector is the glue that tie views to the data model or whatever input context. Using them appropriately is an essential part of the construction of well behaved cubes.

Of course you may have to write your own set of predicates as your needs grows and you get familiar with the framework (see *Defining your own predicates*).

A predicate is a class testing a particular aspect of a context. A selector is built by combining existant predicates or even selectors.

Using and combining existant predicates

You can combine predicates using the `&`, `|` and `~` operators.

When two predicates are combined using the `&` operator, it means that both should return a positive score. On success, the sum of scores is returned.

When two predicates are combined using the `|` operator, it means that one of them should return a positive score. On success, the first positive score is returned.

You can also “negate” a predicate by precedeing it by the `~` unary operator.

Of course you can use parenthesis to balance expressions.

Example

The goal: when on a blog, one wants the RSS link to refer to blog entries, not to the blog entity itself.

To do that, one defines a method on entity classes that returns the RSS stream url for a given entity. The default implementation on `AnyEntity` (the generic entity class used as base for all others) and a specific implementation on `Blog` will do what we want.

But when we have a result set containing several `Blog` entities (or different entities), we don’t know on which entity to call the aforementioned method. In this case, we keep the generic behaviour.

Hence we have two cases here, one for a single-entity rsets, the other for multi-entities rsets.

In `web/views/boxes.py` lies the `RSSIconBox` class. Look at its selector:

```
class RSSIconBox(box.Box):
    """ just display the RSS icon on uniform result set """
    __select__ = box.Box.__select__ & non_final_entity()
```

It takes into account:

- the inherited selection criteria (one has to look them up in the class hierarchy to know the details)
- `non_final_entity`, which filters on result sets containing non final entities (a ‘final entity’ being synonym for entity attributes type, eg `String`, `Int`, etc)

This matches our second case. Hence we have to provide a specific component for the first case:

```
class EntityRSSIconBox(RSSIconBox):
    """just display the RSS icon on uniform result set for a single entity"""
    __select__ = RSSIconBox.__select__ & one_line_rset()
```

Here, one adds the `one_line_rset` predicate, which filters result sets of size 1. Thus, on a result set containing multiple entities, `one_line_rset` makes the `EntityRSSIconBox` class non selectable. However for a result set with one entity, the `EntityRSSIconBox` class will have a higher score than `RSSIconBox`, which is what we wanted.

Of course, once this is done, you have to:

- fill in the call method of `EntityRSSIconBox`
- provide the default implementation of the method returning the RSS stream url on `AnyEntity`
- redefine this method on `Blog`.

When to use selectors?

Selectors are to be used whenever arises the need of dispatching on the shape or content of a result set or whatever else context (value in request form params, authenticated user groups, etc...). That is, almost all the time.

Here is a quick example:

```
class UserLink(component.Component):
    """if the user is the anonymous user, build a link to login else a link
    to the connected user object with a logout link
    """
    __regid__ = 'loggeduserlink'

    def call(self):
        if self._cw.session.anonymous_session:
            # display login link
            ...
        else:
            # display a link to the connected user object with a logout link
            ...
```

The proper way to implement this with *CubicWeb* is to have two different classes sharing the same identifier but with different selectors so you'll get the correct one according to the context.

```
class UserLink(component.Component):
    """display a link to the connected user object with a logout link"""
    __regid__ = 'loggeduserlink'
    __select__ = component.Component.__select__ & authenticated_user()

    def call(self):
        # display useractions and siteactions
        ...

class AnonUserLink(component.Component):
    """build a link to login"""
    __regid__ = 'loggeduserlink'
    __select__ = component.Component.__select__ & anonymous_user()
```

(continues on next page)

(continued from previous page)

```
def call(self):
    # display login link
    ...
```

The big advantage, aside readability once you're familiar with the system, is that your cube becomes much more easily customizable by improving componentization.

Defining your own predicates

You can use the `objectify_predicate` decorator to easily write your own predicates as simple python functions.

In other cases, you can take a look at the following abstract base classes:

- `ExpectedValuePredicate`
- `EClassPredicate`
- `EntityPredicate`

Debugging selection

Once in a while, one needs to understand why a view (or any application object) is, or is not selected appropriately. Looking at which predicates fired (or did not) is the way. The `traced_selection` context manager to help with that, *if you're running your instance in debug mode*.

5.2.4 Base predicates

Here is a description of generic predicates provided by CubicWeb that should suit most of your needs.

Bare predicates

Those predicates are somewhat dumb, which doesn't mean they're not (very) useful.

- `yes`
- `match_kwargs`
- `appobject_selectable`
- `adaptable`
- `configuration_values`

Result set predicates

Those predicates are looking for a result set in the context ('rset' argument or the input context) and match or not according to its shape. Some of these predicates have different behaviour if a particular cell of the result set is specified using 'row' and 'col' arguments of the input context or not.

- `none_rset`
- `any_rset`
- `nonempty_rset`
- `empty_rset`

- *one_line_rset*
- *multi_lines_rset*
- *multi_columns_rset*
- *paginated_rset*
- *sorted_rset*
- *one_etype_rset*
- *multi_etypes_rset*

Entity predicates

Those predicates are looking for either an *entity* argument in the input context, or entity found in the result set ('rset' argument or the input context) and match or not according to entity's (instance or class) properties.

- *non_final_entity*
- *is_instance*
- *score_entity*
- *rql_condition*
- *relation_possible*
- *partial_relation_possible*
- *has_related_entities*
- *partial_has_related_entities*
- *has_permission*
- *has_add_permission*
- *has_mimetype*
- *is_in_state*
- *on_fire_transition*

Logged user predicates

Those predicates are looking for properties of the user issuing the request.

- *match_user_groups*

Web request predicates

Those predicates are looking for properties of *web* request, they can not be used on the data repository side.

- *no_cnx*
- *anonymous_user*
- *authenticated_user*
- *match_form_params*
- *match_search_state*

- `match_context_prop`
- `match_context`
- `match_view`
- `primary_view`
- `contextual`
- `specified_etype_implements`
- `attribute_edited`
- `match_transition`

Other predicates

- `match_exception`
- `debug_mode`

You'll also find some other (very) specific predicates hidden in other modules than `cubicweb.predicates`.

5.3 Data model

This chapter describes how you define a schema and how to make it evolves as the time goes.

5.3.1 Yams schema

The **schema** is the core piece of a *CubicWeb* instance as it defines and handles the data model. It is based on entity types that are either already defined in *Yams* and the *CubicWeb* standard library; or more specific types defined in cubes. The schema for a cube is defined in a *schema* python module or package.

Overview

The core idea of the yams schema is not far from the classical *Entity-relationship* model. But while an E/R model (or *logical model*) traditionally has to be manually translated to a lower-level data description language (such as the SQL *create table* sublanguage), also often described as the *physical model*, no such step is required with *Yams* and *CubicWeb*.

This is because in addition to high-level, logical *Yams* models, one uses the *RQL* data manipulation language to query, insert, update and delete data. *RQL* abstracts as much of the underlying SQL database as a *Yams* schema abstracts from the physical layout. The vagaries of SQL are avoided.

As a bonus point, such abstraction make it quite comfortable to build or use different backends to which *RQL* queries apply.

So, as in the E/R formalism, the building blocks are **entities** (*Entity type*), **relationships** (*Relation type*, *Relation definition*) and **attributes** (handled like relation with *Yams*).

Let us detail a little the divergences between E/R and *Yams*:

- all relationship are binary which means that to represent a non-binary relationship, one has to use an entity,
- relationships do not support attributes (yet, see: <http://www.cubicweb.org/ticket/341318>), hence the need to reify it as an entity if need arises,

- all entities have an *eid* attribute (an integer) that is its primary key (but it is possible to declare uniqueness on other attributes)

Also *Yams* supports the notions of:

- entity inheritance (quite experimental yet, and completely undocumented),
- relation type: that is, relationships can be established over a set of couple of entity types (henre the distinction made between *RelationType* and *RelationDefinition* below)

Finally *Yams* has a few concepts of its own:

- relationships being oriented and binary, we call the left hand entity type the *subject* and the right hand entity type the *object*

Note: The *Yams* schema is available at run time through the `.schema` attribute of the *vregistry*. It's an instance of `cubicweb.schema.Schema`, which extends `yams.schema.Schema`.

Entity type

An entity type is an instance of `yams.schema.EntitySchema`. Each entity type has a set of attributes and relations, and some permissions which define who can add, read, update or delete entities of this type.

The following built-in types are available: `String`, `Int`, `BigInt`, `Float`, `Decimal`, `Boolean`, `Date`, `Datetime`, `Time`, `Interval`, `Byte` and `Password`. They can only be used as attributes of an other entity type.

There is also a *RichString* kindof type:

```
yams.buildobjs.RichString(default_format: str = 'text/plain', format_constraints:
                           typing.Optional[typing.List[yams.constraints.BaseConstraint]] = None, required:
                           bool = False, maxsize: typing.Optional[int] = None, formula=<nullobject>,
                           vocabulary: typing.Optional[typing.List[str]] = None, unique:
                           typing.Optional[bool] = None, override: bool = False, **kwargs)
```

`RichString` is a convenience attribute type for attribute containing text in a format that should be specified in another attribute.

The following declaration:

```
class Card(EntityType):
    content = RichString(fulltextindexed=True, default_format='text/rest')
```

is equivalent to:

```
class Card(EntityType):
    content_format = String(internationalizable=True,
                           default='text/rest', constraints=[FORMAT_CONSTRAINT])
    content = String(fulltextindexed=True)
```

The `__unique_together__` class attribute is a list of tuples of names of attributes or inlined relations. For each tuple, CubicWeb ensures the unicity of the combination. For example:

```
class State(EntityType):
    __unique_together__ = [('name', 'state_of')]

    name = String(required=True)
```

(continues on next page)

(continued from previous page)

```
state_of = SubjectRelation('Workflow', cardinality='1*',
                           composite='object', inlined=True)
```

You can find more base entity types in *Pre-defined entities in the library*.

Relation type

A relation type is an instance of `yams.schema.RelationSchema`. A relation type is simply a semantic definition of a kind of relationship that may occur in an application.

It may be referenced by zero, one or more relation definitions.

It is important to choose a good name, at least to avoid conflicts with some semantically different relation defined in other cubes (since there's only a shared name space for these names).

A relation type holds the following properties (which are hence shared between all relation definitions of that type):

- *inlined*: boolean handling the physical optimization for archiving the relation in the subject entity table, instead of creating a specific table for the relation. This applies to relations where cardinality of subject->relation->object is 0..1 (?) or 1..1 (I) for *all* its relation definitions.
- *symmetric*: boolean indicating that the relation is symmetrical, which means that *X relation Y* implies *Y relation X*.

Relation definition

A relation definition is an instance of `yams.schema.RelationDefinition`. It is a complete triplet “<subject entity type> <relation type> <object entity type>”.

When creating a new instance of that class, the corresponding `RelationType` instance is created on the fly if necessary.

Properties

The available properties for relation definitions are enumerated here. There are several kind of properties, as some relation definitions are actually attribute definitions, and other are not.

Some properties may be completely optional, other may have a default value.

Common properties for attributes and relations:

- *description*: a unicode string describing an attribute or a relation. By default this string will be used in the editing form of the entity, which means that it is supposed to help the end-user and should be flagged by the function `_` to be properly internationalized.
- *constraints*: a list of conditions/constraints that the relation has to satisfy (c.f. [Constraints](#))
- *cardinality*: a two character string specifying the cardinality of the relation. The first character defines the cardinality of the relation on the subject, and the second on the object. When a relation can have multiple subjects or objects, the cardinality applies to all, not on a one-to-one basis (so it must be consistent...). Default value is ‘**’. The possible values are inspired from regular expression syntax:
 - I: 1..1
 - ?: 0..1
 - +: 1..n
 - *: 0..n

Attributes properties:

- *unique*: boolean indicating if the value of the attribute has to be unique or not within all entities of the same type (false by default)
- *indexed*: boolean indicating if an index needs to be created for this attribute in the database (false by default). This is useful only if you know that you will have to run numerous searches on the value of this attribute.
- *default*: default value of the attribute. In case of date types, the values which could be used correspond to the RQL keywords *TODAY* and *NOW*.
- *metadata*: Is also accepted as an argument of the attribute constructor. It is not really an attribute property. see [Metadata](#) for details.

Properties for *String* attributes:

- *fulltextindexed*: boolean indicating if the attribute is part of the full text index (false by default) (*applicable on the type 'Byte' as well*)
- *internationalizable*: boolean indicating if the value of the attribute is internationalizable (false by default)

Relation properties:

- *composite*: string indicating that the subject (*composite* == 'subject') is composed of the objects of the relations. For the opposite case (when the object is composed of the subjects of the relation), we just set 'object' as value. The composition implies that when the relation is deleted (so when the composite is deleted, at least), the composed are also deleted.
- *fulltext_container*: string indicating if the value if the full text indexation of the entity on one end of the relation should be used to find the entity on the other end. The possible values are 'subject' or 'object'. For instance the *use_email* relation has that property set to 'subject', since when performing a full text search people want to find the entity using an email address, and not the entity representing the email address.

Constraints

By default, the available constraint types are:

General Constraints

- *SizeConstraint*: allows to specify a minimum and/or maximum size on string (generic case of *maxsize*)
- *BoundaryConstraint*: allows to specify a minimum and/or maximum value on numeric types and date

```
from yams.constraints import BoundaryConstraint, TODAY, NOW, Attribute

class DatedEntity(EntityType):
    start = Date(constraints=[BoundaryConstraint('>=', TODAY())])
    end = Date(constraints=[BoundaryConstraint('>=', Attribute('start'))])

class Before(EntityType):
    last_time = DateTime(constraints=[BoundaryConstraint('<=', NOW())])
```

- *IntervalBoundConstraint*: allows to specify an interval with included values

```
class Node(EntityType):
    latitude = Float(constraints=[IntervalBoundConstraint(-90, +90)])
```

- *UniqueConstraint*: identical to "unique=True"

- *StaticVocabularyConstraint*: identical to “vocabulary=(...)”
- *RegexConstraint*: set allowed patterns as a regexp

```
from yams.constraints import RegexConstraint

class Organisation(EntityType):
    name = String(constraints=[RegexConstraint('^[^_]*$')])
```

Constraints can be dependent on a fixed value (90, Date(2015,3,23)) or a variable. In this second case, yams can handle :

- *Attribute*: compare to the value of another attribute.
- *TODAY*: compare to the current Date.
- *NOW*: compare to the current Datetime.

See [YAMS constraint module documentation](#).

RQL Based Constraints

RQL based constraints may take three arguments. The first one is the **WHERE** clause of a RQL query used by the constraint. The second argument **mainvars** is the **Any** clause of the query. By default this include *S* reserved for the subject of the relation and *O* for the object. Additional variables could be specified using **mainvars**. The argument expects a single string with all variable's name separated by spaces. The last one, **msg**, is the error message displayed when the constraint fails. As *RQLVocabularyConstraint* never fails the third argument is not available.

- *RQLConstraint*: allows to specify a RQL query that has to be satisfied by the subject and/or the object of relation. In this query the variables *S* and *O* are reserved for the relation subject and object entities.
- *RQLVocabularyConstraint*: similar to the previous type of constraint except that it does not express a “strong” constraint, which means it is only used to restrict the values listed in the drop-down menu of editing form, but it does not prevent another entity to be selected.
- *RQLUniqueConstraint*: allows to the specify a RQL query that ensure that an attribute is unique in a specific context. The Query must **never** return more than a single result to be satisfied. In this query the variables *S* is reserved for the relation subject entity. The other variables should be specified with the second constructor argument (**mainvars**). This constraint type should be used when `__unique_together__` doesn't fit.

The security model

The security model of *CubicWeb* is based on *Access Control List*. The main principles are:

- users and groups of users
- a user belongs to at least one group of user
- permissions (*read*, *update*, *create*, *delete*)
- permissions are assigned to groups (and not to users)

For *CubicWeb* in particular:

- we associate rights at the entities/relations schema level
- the default groups are: *managers*, *users* and *guests*
- users belong to the *users* group
- there is a virtual group called *owners* to which we can associate only *delete* and *update* permissions

- we can not add users to the *owners* group, they are implicitly added to it according to the context of the objects they own
- the permissions of this group are only checked on *update/delete* actions if all the other groups the user belongs to do not provide those permissions

Setting permissions is done with the class attribute `__permissions__` of entity types and relation definitions. The value of this attribute is a dictionary where the keys are the access types (action), and the values are the authorized groups or rql expressions.

For an entity type, the possible actions are *read*, *add*, *update* and *delete*.

For a relation, the possible actions are *read*, *add*, and *delete*.

For an attribute, the possible actions are *read*, *add* and *update*, and they are a refinement of an entity type permission.

Note: By default, the permissions of an entity type attributes are equivalent to the permissions of the entity type itself.

It is possible to provide custom attribute permissions which are stronger than, or are more lenient than the entity type permissions.

In a situation where all attributes were given custom permissions, the entity type permissions would not be checked, except for the *delete* action.

For each access type, a tuple indicates the name of the authorized groups and/or one or multiple RQL expressions to satisfy to grant access. The access is provided if the user is in one of the listed groups or if one of the RQL condition is satisfied.

Default permissions

The default permissions for `EntityType` are:

```
__permissions__ = {
    'read': ('managers', 'users', 'guests',),
    'update': ('managers', 'owners',),
    'delete': ('managers', 'owners'),
    'add': ('managers', 'users',)
}
```

The default permissions for relations are:

```
__permissions__ = {'read': ('managers', 'users', 'guests',),
                  'delete': ('managers', 'users'),
                  'add': ('managers', 'users',),}
```

The default permissions for attributes are:

```
__permissions__ = {'read': ('managers', 'users', 'guests',),
                  'add': ('managers', ERQLEExpression('U has_add_permission X')),
                  'update': ('managers', ERQLEExpression('U has_update_permission X')),}
```

Note: The default permissions for attributes are not syntactically equivalent to the default permissions of the entity types, but the rql expressions work by delegating to the entity type permissions.

The standard user groups

- *guests*
- *users*
- *managers*
- *owners*: virtual group corresponding to the entity's owner. This can only be used for the actions *update* and *delete* of an entity type.

It is also possible to use specific groups if they are defined in the precreate script of the cube (`migration/precreate.py`). Defining groups in postcreate script or later makes them unavailable for security purposes (in this case, an `sync_schema_perms_perms` command has to be issued in a CubicWeb shell).

Use of RQL expression for write permissions

It is possible to define RQL expression to provide update permission (*add*, *delete* and *update*) on entity type / relation definitions. An rql expression is a piece of query (corresponds to the WHERE statement of an RQL query), and the expression will be considered as satisfied if it returns some results. They can not be used in *read* permission.

To use RQL expression in entity type permission:

- you have to use the class `ERQLEExpression`
- in this expression, the variables *X* and *U* are pre-defined references respectively on the current entity (on which the action is verified) and on the user who send the request

For RQL expressions on a relation type, the principles are the same except for the following:

- you have to use the class `RRQLEExpression` instead of `ERQLEExpression`
- in the expression, the variables *S*, *O* and *U* are pre-defined references to respectively the subject and the object of the current relation (on which the action is being verified) and the user who executed the query

To define security for attributes of an entity (non-final relation), you have to use the class `ERQLEExpression` in which *X* represents the entity the attribute belongs to.

It is possible to use in those expression a special relation *has_<ACTION>_permission* where the subject is the user (eg 'U') and the object is any variable representing an entity (usually 'X' in `ERQLEExpression`, 'S' or 'O' in `RRQLEExpression`), meaning that the user needs to have permission to execute the action <ACTION> on the entities represented by this variable. It's recommended to use this feature whenever possible since it simplify greatly complex security definition and upgrade.

```
class my_relation(RelationDefinition):
    __permissions__ = {'read': ('managers', 'users'),
                      'add': ('managers', RRQLEExpression('U has_update_permission S')),
                      'delete': ('managers', RRQLEExpression('U has_update_permission S')),
                      }
```

In the above example, user will be allowed to add/delete *my_relation* if he has the *update* permission on the subject of the relation.

Note: Potentially, the *use of an RQL expression to add an entity or a relation* can cause problems for the user interface, because if the expression uses the entity or the relation to create, we are not able to verify the permissions before we actually added the entity (please note that this is not a problem for the RQL server at all, because the permissions checks are done after the creation). In such case, the permission check methods (`CubicWebEntitySchema.check_perm` and `has_perm`) can indicate that the user is not allowed to create this entity while it would obtain the permission. To

compensate this problem, it is usually necessary in such case to use an action that reflects the schema permissions but which check properly the permissions so that it would show up only if possible.

Use of RQL expression for reading rights

The principles are the same but with the following restrictions:

- you can not use rql expression for the *read* permission of relations and attributes,
- you can not use special *has_<ACTION>_permission* relation in the rql expression.

Important notes about write permissions checking

Write permissions (e.g. ‘add’, ‘update’, ‘delete’) are checked in core hooks.

When a permission is checked slightly vary according to if it’s an entity or relation, and if the relation is an attribute relation or not). It’s important to understand that since according to when a permission is checked, values returned by rql expressions may changes, hence the permission being granted or not.

Here are the current rules:

1. permission to add/update entity and its attributes are checked on commit
2. permission to delete an entity is checked in ‘before_delete_entity’ hook
3. permission to add a relation is checked either:
 - in ‘before_add_relation’ hook if the relation type is in the *BEFORE_ADD_RELATIONS* set
 - else at commit time if the relation type is in the *ON_COMMIT_ADD_RELATIONS* set
 - else in ‘after_add_relation’ hook (the default)
4. permission to delete a relation is checked in ‘before_delete_relation’ hook

Last but not least, remember queries issued from hooks and operation are by default ‘unsafe’, eg there are no read or write security checks.

See `cubicweb.hooks.security` for more details.

5.3.2 Derived attributes and relations

Note: **TODO** Check organisation of the whole chapter of the documentation

Cubicweb offers the possibility to *query* data using so called *computed* relations and attributes. Those are *seen* by RQL requests as normal attributes and relations but are actually derived from other attributes and relations. In a first section we’ll informally review two typical use cases. Then we see how to use computed attributes and relations in your schema. Last we will consider various significant aspects of their implementation and the impact on their usage.

Motivating use cases

Computed (or reified) relations

It often arises that one must represent a ternary relation, or a family of relations. For example, in the context of an exhibition catalog you might want to link all *contributors* to the *work* they contributed to, but this contribution can be as *illustrator*, *author*, *performer*, ...

The classical way to describe this kind of information within an entity-relationship schema is to *reify* the relation, that is turn the relation into an entity. In our example the schema will have a *Contribution* entity type used to represent the family of the contribution relations.

```
class ArtWork(EntityType):
    name = String()
    ...

class Person(EntityType):
    name = String()
    ...

class Contribution(EntityType):
    contributor = SubjectRelation('Person', cardinality='1*', inlined=True)
    manifestation = SubjectRelation('ArtWork')
    role = SubjectRelation('Role')

class Role(EntityType):
    name = String()
```

But then, in order to query the illustrator(s) I of a work W, one has to write:

```
Any I, W WHERE C is Contribution, C contributor I, C manifestation W,
               C role R, R name 'illustrator'
```

whereas we would like to be able to simply write:

```
Any I, W WHERE I illustrator_of W
```

This is precisely what the computed relations allow.

Computed (or synthesized) attribute

Assuming a trivial schema for describing employees in companies, one can be interested in the total of salaries paid by a company for all its employees. One has to write:

```
Any C, SUM(SA) GROUPBY S WHERE E works_for C, E salary SA
```

whereas it would be most convenient to simply write:

```
Any C, TS WHERE C total_salary TS
```

And this is again what computed attributes provide.

Using computed attributes and relations

Computed (or reified) relations

In the above case we would define the *computed relation* `illustrator_of` in the schema by:

```
class illustrator_of(ComputedRelation):  
    rule = ('C is Contribution, C contributor S, C manifestation O,'  
           'C role R, R name "illustrator"')
```

You will note that:

- the S and O RQL variables implicitly identify the subject and object of the defined computed relation, akin to what happens in `RRQLExpression`
- the possible subject and object entity types are inferred from the rule;
- computed relation definitions always have empty *add* and *delete* permissions
- *read* permissions can be defined, permissions from the relations used in the rewrite rule **are not considered** ;
- nothing else may be defined on the *ComputedRelation* subclass beside description, permissions and rule (e.g. no cardinality, composite, etc.). *BadSchemaDefinition* is raised on attempt to specify other attributes;
- computed relations can not be used in ‘SET’ and ‘DELETE’ rql queries (*BadQuery* exception raised).

NB: The fact that the *add* and *delete* permissions are *empty* even for managers is expected to make the automatic UI not attempt to edit them.

Computed (or synthesized) attributes

In the above case we would define the *computed attribute* `total_salary` on the `Company` entity type in the schema by:

```
class Company(EntityType):  
    name = String()  
    total_salary = Int(formula='Any SUM(SA) GROUPBY E WHERE P works_for X, E salary SA')
```

- the X RQL variable implicitly identifies the entity holding the computed attribute, akin to what happens in `ERQL-Expression`;
- the type inferred from the formula is checked against the declared type, and *BadSchemaDefinition* is raised if they don’t match;
- the computed attributes always have empty *update* permissions
- *BadSchemaDefinition* is raised on attempt to set ‘update’ permissions;
- ‘read’ permissions can be defined, permissions regarding the formula **are not considered**;
- other attribute’s property (inlined, ...) can be defined as for normal attributes;
- Similarly to computed relation, computed attribute can’t be used in ‘SET’ and ‘DELETE’ rql queries (*BadQuery* exception raised).

API and implementation

Representation in the data backend

Computed relations have no direct representation at the SQL table level. Instead, each time a query is issued the query is rewritten to replace the computed relation by its equivalent definition and the resulting rewritten query is performed in the usual way.

On the contrary, computed attributes are represented as a column in the table for their host entity type, just like normal attributes. Their value is kept up-to-date with respect to their definition by a system of hooks (also called triggers in most RDBMS) which recomputes them when the relations and attributes they depend on are modified.

Yams API

When accessing the schema through the *yams API* (not when defining a schema in a `schema.py` file) the computed attributes and relations are represented as follows:

relations The `yams.RelationSchema` class has a new `rule` attribute holding the rule as a string. If this attribute is set all others must not be set.

attributes A new property `formula` is added on class `yams.RelationDefinitionSchema` along with a new keyword argument `formula` on the initializer.

Migration

The migrations are to be handled as summarized in the array below.

	Computed rtype	Computed attribute
add	<ul style="list-style-type: none"> • <code>add_relation_type</code> • <code>add_relation_definition</code> should trigger an error 	<ul style="list-style-type: none"> • <code>add_attribute</code> • <code>add_relation_definition</code>
modify (rule or formula)	<ul style="list-style-type: none"> • <code>sync_schema_prop_perms</code>: checks the rule is synchronized with the database 	<ul style="list-style-type: none"> • <code>sync_schema_prop_perms</code>: <ul style="list-style-type: none"> – empty the cache, – check formula, – make sure all the values get updated
del	<ul style="list-style-type: none"> • <code>drop_relation_type</code> • <code>drop_relation_definition</code> should trigger an error 	<ul style="list-style-type: none"> • <code>drop_attribute</code> • <code>drop_relation_definition</code>

5.3.3 Defining your schema using yams

Entity type definition

An entity type is defined by a Python class which inherits from `yams.buildobjs.EntityType`. The class definition contains the description of attributes and relations for the defined entity type. The class name corresponds to the entity type name. It is expected to be defined in the module `mycube.schema`.

Note on schema definition The code in `mycube.schema` is not meant to be executed. The class `EntityType` mentioned above is different from the `EntitySchema` class described in the previous chapter. `EntityType` is a helper class to make Entity definition easier. Yams will process `EntityType` classes and create `EntitySchema` instances from these class definitions. Similar manipulation happen for relations.

When defining a schema using python files, you may use the following shortcuts:

- *required*: boolean indicating if the attribute is required, ed subject cardinality is ‘1’
- *vocabulary*: specify static possible values of an attribute
- *maxsize*: integer providing the maximum size of a string (no limit by default)

For example:

```
class Person(EntityType):
    """A person with the properties and the relations necessary for my
    application"""

    last_name = String(required=True, fulltextindexed=True)
    first_name = String(required=True, fulltextindexed=True)
    title = String(vocabulary=('Mr', 'Mrs', 'Miss'))
    date_of_birth = Date()
    works_for = SubjectRelation('Company', cardinality='?*')
```

The entity described above defines three attributes of type `String`, `last_name`, `first_name` and `title`, an attribute of type `Date` for the date of birth and a relation that connects a *Person* to another entity of type *Company* through the semantic *works_for*.

Naming convention Entity class names must start with an uppercase letter. The common usage is to use CamelCase names.

Attribute and relation names must start with a lowercase letter. The common usage is to use `underscore_separated_words`. Attribute and relation names starting with a single underscore are permitted, to denote a somewhat “protected” or “private” attribute.

In any case, identifiers starting with “CW” or “cw” are reserved for internal use by the framework.

Some attribute using the name of another attribute as prefix are considered metadata. For example, if an `EntityType` have both a `data` and `data_format` attribute, `data_format` is view as the `format` metadata of `data`. Later the `cw_attr_metadata()` method will allow you to fetch metadata related to an attribute. There are only three valid metadata names: `format`, `encoding` and `name`.

The name of the Python attribute corresponds to the name of the attribute or the relation in *CubicWeb* application.

An attribute is defined in the schema as follows:

```
attr_name = AttrType(*properties, metadata={})
```

where

- *AttrType*: is one of the type listed in *EntityType*,

- *properties*: is a list of the attribute needs to satisfy (see [Properties](#) for more details),
- *metadata*: is a dictionary of meta attributes related to `attr_name`. Dictionary keys are the name of the meta attribute. Dictionary values attributes objects (like the content of `AttrType`). For each entry of the metadata dictionary a `<attr_name>_<key> = <value>` attribute is automatically added to the `EntityType`. see [Metadata](#) section for details about valid key.

—

While building your schema

- it is possible to use the attribute *meta* to flag an entity type as a *meta* (e.g. used to describe/categorize other entities)

Note: if you end up with an *if* in the definition of your entity, this probably means that you need two separate entities that implement the *ITree* interface and get the result from `.children()` which ever entity is concerned.

Definition of relations

A relation is defined by a Python class heriting *RelationType*. The name of the class corresponds to the name of the type. The class then contains a description of the properties of this type of relation, and could as well contain a string for the subject and a string for the object. This allows to create new definition of associated relations, (so that the class can have the definition properties from the relation) for example

```
class locked_by(RelationType):
    """relation on all entities indicating that they are locked"""
    inlined = True
    cardinality = '?*'
    subject = '*'
    object = 'CWUser'
```

If provided, the *subject* and *object* attributes denote the subject and object of the various relation definitions related to the relation type. Allowed values for these attributes are:

- a string corresponding to an entity type
- a tuple of string corresponding to multiple entity types
- the '*' special string, meaning all types of entities

When a relation is not inlined and not symmetrical, and it does not require specific permissions, it can be defined using a *SubjectRelation* attribute in the *EntityType* class. The first argument of *SubjectRelation* gives the entity type for the object of the relation.

Naming convention Although this way of defining relations uses a Python class, the naming convention defined earlier prevails over the PEP8 conventions used in the framework: relation type class names use `underscore_separated_words`.

Historical note It has been historically possible to use *ObjectRelation* which defines a relation in the opposite direction. This feature is deprecated and therefore should not be used in newly written code.

Future deprecation note In an even more remote future, it is quite possible that the *SubjectRelation* shortcut will become deprecated, in favor of the *RelationType* declaration which offers some advantages in the context of reusable cubes.

Handling schema changes

Also, it should be clear that to properly handle data migration, an instance's schema is stored in the database, so the python schema file used to defined it is only read when the instance is created or upgraded.

5.3.4 Metadata

Each entity type in *CubicWeb* has at least the following meta-data attributes and relations:

eid entity's identifier which is unique in an instance. We usually call this identifier *eid* for historical reason.

creation_date Date and time of the creation of the entity.

modification_date Date and time of the latest modification of an entity.

cwuri Reference URL of the entity, which is not expected to change.

created_by Relation to the *users* who has created the entity

owned_by Relation to *users* whom the entity belongs; usually the creator but not necessary, and it could have multiple owners notably for permission control

is Relation to the *entity type* of which type the entity is.

is_instance Relation to the *entity types* of which type the entity is an instance of.

5.3.5 Pre-defined entities in the library

The library defines a set of entity schemas that are required by the system or commonly used in *CubicWeb* instances.

Entity types used to store the schema

- CWEType, entity type
- CWRTType, relation type
- CWRelation, relation definition
- CWAttribute, attribute relation definition
- CWConstraint, *CWConstraintType*, *RQLEExpression*

Entity types used to manage users and permissions

- CWUser, system users
- CWGroup, users groups

Entity types used to manage workflows

- *Workflow*, workflow entity, linked to some entity types which may use this workflow
- State, workflow state
- Transition, workflow transition
- TrInfo, record of a transition traffic for an entity

Other entity types

- CWProperty, used to configure the instance
- EmailAddress, email address, used by the system to send notifications to the users and also used by others optionnals schemas
- Bookmark, an entity type used to allow a user to customize his links within the instance
- ExternalUri, used for semantic web site to indicate that an entity is the same as another from an external site

5.3.6 Defining a Workflow

General

A workflow describes how certain entities have to evolve between different states. Hence we have a set of states, and a “transition graph”, i.e. a set of possible transitions from one state to another state.

We will define a simple workflow for a blog, with only the following two states: *submitted* and *published*. You may want to take a look at *Building a simple blog with CubicWeb* if you want to quickly setup an instance running a blog.

Setting up a workflow

We want to create a workflow to control the quality of the BlogEntry submitted on the instance. When a BlogEntry is created by a user its state should be *submitted*. To be visible to all, it has to be in the state *published*. To move it from *submitted* to *published*, we need a transition that we can call *approve_blogentry*.

A BlogEntry state should not be modifiable by every user. So we have to define a group of users, *moderators*, and this group will have appropriate permissions to publish a BlogEntry.

There are two ways to create a workflow: from the user interface, or by defining it in `migration/postcreate.py`. This script is executed each time a new `cubicweb-ctl db-init` is done. We strongly recommend to create the workflow in `migration/postcreate.py` and we will now show you how. Read *Two bits of warning* to understand why.

The state of an entity is managed by the `in_state` attribute which can be added to your entity schema by inheriting from `cubicweb.schema.WorkflowableEntityType`.

About our example of BlogEntry, we must have:

```
from cubicweb.schema import WorkflowableEntityType

class BlogEntry(WorkflowableEntityType):
    ...
```

Creating states, transitions and group permissions

The `postcreate` script is executed in a special environment, adding several *CubicWeb* primitives that can be used.

They are all defined in the `ServerMigrationHelper` class. We will only discuss the methods we use to create a workflow in this example.

A workflow is a collection of entities of type `State` and of type `Transition` which are standard *CubicWeb* entity types.

To define a workflow for `BlogDemo`, please add the following lines to `migration/postcreate.py`:

```
from cubicweb import _

moderators = add_entity('CWGroup', name=u"moderators")
```

This adds the *moderators* user group.

```
wf = add_workflow(u'blog publication workflow', 'BlogEntry')
```

At first, instantiate a new workflow object with a gentle description and the concerned entity types (this one can be a tuple for multiple value).

```
submitted = wf.add_state(_('submitted'), initial=True)
published = wf.add_state(_('published'))
```

This will create two entities of type `State`, one with name 'submitted', and the other with name 'published'.

`add_state` expects as first argument the name of the state you want to create and an optional argument to say if it is supposed to be the initial state of the entity type.

```
wf.add_transition(_('approve_blogentry'), (submitted,), published, ('moderators',
→ 'managers'),)
```

This will create an entity of type `Transition` with name *approve_blogentry* which will be linked to the `State` entities created before.

`add_transition` expects

- as the first argument: the name of the transition
- then the list of states on which the transition can be triggered,
- the target state of the transition,
- and the permissions (e.g. a list of user groups who can apply the transition; the user has to belong to at least one of the listed group to perform the action).

Note: Do not forget to add the `_()` in front of all states and transitions names while creating a workflow so that they will be identified by the i18n catalog scripts.

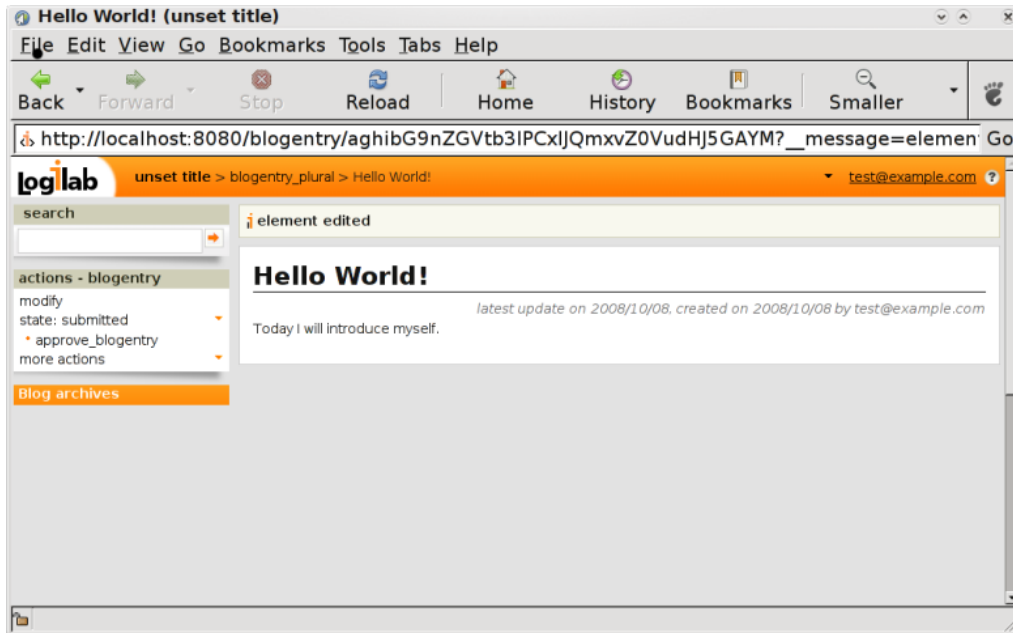
In addition to the user groups (one of which the user needs to belong to), we could have added a RQL condition. In this case, the user can only perform the action if the two conditions are satisfied.

If we use an RQL condition on a transition, we can use the following variables:

- `X`, the entity on which we may pass the transition

- *U*, the user executing that may pass the transition

It's also possible to get a given transition (usefull in migration) from a workflow use *transition_by_name(trname)*. To update the permission associated to the transition use *set_permissions(requiredgroups=(), conditions=(), reset=True)*. If *reset* is False, then the new permission are added instead of replacing the old one.



You can notice that in the action box of a BlogEntry, the state is now listed as well as the possible transitions for the current state defined by the workflow.

The transitions will only be displayed for users having the right permissions. In our example, the transition *approve_blogentry* will only be displayed for the users belonging to the group *moderators* or *managers*.

Two bits of warning

We could perfectly use the administration interface to do these operations. It is a convenient thing to do at times (when doing development, to quick-check things). But it is not recommended beyond that because it is a bit complicated to do it right and it will be only local to your instance (or, said a bit differently, such a workflow only exists in an instance database). Furthermore, you cannot write unit tests against deployed instances, and experience shows it is mandatory to have tests for any mildly complicated workflow setup.

Indeed, if you create the states and transitions through the user interface, next time you initialize the database you will have to re-create all the workflow entities. The user interface should only be a reference for you to view the states and transitions, but is not the appropriate interface to define your application workflow.

Alternative way to declare workflows

Workflow setup utilities.

These functions work with a declarative workflow definition:

```
{
  'etypes': 'CWGroup',
  'default': True,
  'initial_state': u'draft',
  'states': [u'draft', u'published'],
  'transitions': {
    u'publish': {
      'fromstates': u'draft',
      'tostate': u'published',
      'requiredgroups': u'managers'
      'conditions': (
        'U in_group X',
        'X owned_by U'
      )
    }
  }
}
```

`cubicweb.wfutils.setup_workflow(cnx, name, wfdef, cleanup=True)`

Create or update a workflow definition so it matches the given definition.

Parameters

- **cnx** – A connexion with enough permissions to define a workflow
- **name** – The workflow name. Used to create the *Workflow* entity, or to find an existing one.
- **wfdef** – A workflow definition.
- **cleanup** – Remove extra states and transitions. Can be done separately by calling `cleanupworkflow()`.

Returns The created/updated workflow entity

`cubicweb.wfutils.cleanupworkflow(cnx, wf, wfdef)`

Cleanup an existing workflow by removing the states and transitions that do not exist in the given definition.

Parameters

- **cnx** – A connexion with enough permissions to define a workflow
- **wf** – A *Workflow* entity
- **wfdef** – A workflow definition

5.4 Data as objects

In this chapter, we will introduce the objects that are used to handle the logic associated to the data stored in the database.

5.4.1 Access to persistent data

Python-level access to persistent data is provided by the `Entity` class.

An entity class is bound to a schema entity type. Descriptors are added when classes are registered in order to initialize the class according to its schema:

- the attributes defined in the schema appear as attributes of these classes
- the relations defined in the schema appear as attributes of these classes, but are lists of instances

Formatting and output generation:

- `view(__vid, __registry='views', **kwargs)()`, applies the given view to the entity (and returns a uni-code string)
- `absolute_url(*args, **kwargs)()`, returns an absolute URL including the base-url
- `rest_path()`, returns a relative REST URL to get the entity
- `printable_value(attr, value=_marker, attrtype=None, format='text/html', displaytime=True)()`, returns a string enabling the display of an attribute value in a given format (the value is automatically recovered if necessary)

Data handling:

- `as_rset()`, converts the entity into an equivalent result set simulating the request *Any X WHERE X eid _eid_*
- `complete(skip_bytes=True)()`, executes a request that recovers at once all the missing attributes of an entity
- `get_value(name)()`, returns the value associated to the attribute name given in parameter
- `related(rtype, role='subject', limit=None, entities=False)()`, returns a list of entities related to the current entity by the relation given in parameter
- `unrelated(rtype, targettype, role='subject', limit=None)()`, returns a result set corresponding to the entities not (yet) related to the current entity by the relation given in parameter and satisfying its constraints
- `cw_set(**kwargs)()`, updates entity's attributes and/or relation with the corresponding values given named parameters. To set a relation where this entity is the object of the relation, use *reverse_<relation>* as argument name. Values may be an entity, a list of entities, or None (meaning that all relations of the given type from or to this object should be deleted).
- `copy_relations(ceid)()`, copies the relations of the entities having the eid given in the parameters on the current entity
- `cw_delete()` allows to delete the entity

5.4.2 The `AnyEntity` class

To provide a specific behavior for each entity, we can define a class inheriting from `cubicweb.entities.AnyEntity`. In general, we define this class in `mycube.entities` module (or in a submodule if we want to split code among multiple files) so that it will be available on both server and client side.

The class `AnyEntity` is a sub-class of `Entity` that add methods to it, and helps specializing (by further subclassing) the handling of a given entity type.

Most methods defined for `AnyEntity`, in addition to `Entity`, add support for the [Dublin Core](#) metadata.

Standard meta-data (Dublin Core):

- `dc_title()`, returns a unicode string corresponding to the meta-data *Title* (used by default is the first non-meta attribute of the entity schema)
- `dc_long_title()`, same as `dc_title` but can return a more detailed title
- `dc_description(format='text/plain')()`, returns a unicode string corresponding to the meta-data *Description* (looks for a description attribute by default)
- `dc_authors()`, returns a unicode string corresponding to the meta-data *Authors* (owners by default)
- `dc_creator()`, returns a unicode string corresponding to the creator of the entity
- `dc_date(date_format=None)()`, returns a unicode string corresponding to the meta-data *Date* (update date by default)
- `dc_type(form='')()`, returns a string to display the entity type by specifying the preferred form (*plural* for a plural form)
- `dc_language()`, returns the language used by the entity

5.4.3 Inheritance

When describing a data model, entities can inherit from other entities as is common in object-oriented programming.

You have the possibility to redefine whatever pleases you, as follow:

```
from cubicweb_OTHER_CUBE import entities

class EntityExample(entities.EntityExample):

    def dc_long_title(self):
        return '%s (%s)' % (self.name, self.description)
```

The most specific entity definition will always be the one used by the ORM. For instance, the new `EntityExample` above in `mycube` replaces the one in `OTHER_CUBE`. These types are stored in the *etype* section of the *vregistry*.

Notice this is different than yams schema inheritance, which is an experimental undocumented feature.

5.4.4 Application logic

While a lot of custom behaviour and application logic can be implemented using entity classes, the programmer must be aware that adding new attributes and method on an entity class adds may shadow schema-level attribute or relation definitions.

To keep entities clean (mostly data structures plus a few universal methods such as listed above), one should use *adapters* (see *Interfaces and Adapters*).

5.4.5 Loaded attributes and default sorting management

- The class attribute *fetch_attrs* allows to define in an entity class a list of names of attributes that should be automatically loaded when entities of this type are fetched from the database using ORM methods retrieving entity of this type (such as `related()` and `unrelated()`). You can also put relation names in there, but we are limited to *subject relations of cardinality '?' or '1'*.
- The `cw_fetch_order()` and `cw_fetch_unrelated_order()` class methods are respectively responsible to control how entities will be sorted when:
 - retrieving all entities of a given type, or entities related to another
 - retrieving a list of entities for use in drop-down lists enabling relations creation in the editing view of an entity

By default entities will be listed on their modification date descending, i.e. you'll get entities recently modified first. While this is usually a good default in drop-down list, you'll probably want to change `cw_fetch_order`.

This may easily be done using the `fetch_config()` function, which simplifies the definition of attributes to load and sorting by returning a list of attributes to pre-load (considering automatically the attributes of *AnyEntity*) and a sorting function as described below:

```
cubicweb.entities.fetch_config(fetchattrs, mainattr=None, pclass=<class 'cubicweb.entities.AnyEntity'>,
                               order='ASC')
```

function to ease basic configuration of an entity class ORM. Basic usage is:

```
class MyEntity(AnyEntity):

    fetch_attrs, cw_fetch_order = fetch_config(['attr1', 'attr2'])
    # uncomment line below if you want the same sorting for 'unrelated' entities
    # cw_fetch_unrelated_order = cw_fetch_order
```

Using this, when using ORM methods retrieving this type of entity, 'attr1' and 'attr2' will be automatically prefetched and results will be sorted on 'attr1' ascending (ie the first attribute in the list).

This function will automatically add to fetched attributes those defined in parent class given using the *pclass* argument.

Also, You can use *mainattr* and *order* argument to have a different sorting.

In you want something else (such as sorting on the result of a registered procedure), here is the prototype of those methods:

```
classmethod Entity.cw_fetch_order(select, attr, var)
```

This class method may be used to control sort order when multiple entities of this type are fetched through ORM methods. Its arguments are:

- *select*, the RQL syntax tree
- *attr*, the attribute being watched

- *var*, the variable through which this attribute's value may be accessed in the query

When you want to do some sorting on the given attribute, you should modify the syntax tree accordingly. For instance:

```
from rql import nodes

class Version(AnyEntity):
    __regid__ = 'Version'

    fetch_attrs = ('num', 'description', 'in_state')

    @classmethod
    def cw_fetch_order(cls, select, attr, var):
        if attr == 'num':
            func = nodes.Function('version_sort_value')
            func.append(nodes.variable_ref(var))
            sterm = nodes.SortTerm(func, asc=False)
            select.add_sort_term(sterm)
```

The default implementation call `cw_fetch_unrelated_order()`

classmethod `Entity.cw_fetch_unrelated_order(select, attr, var)`

This class method may be used to control sort order when multiple entities of this type are fetched to use in edition (e.g. propose them to create a new relation on an edited entity).

See `cw_fetch_unrelated_order()` for a description of its arguments and usage.

By default entities will be listed on their modification date descending, i.e. you'll get entities recently modified first.

5.4.6 Interfaces and Adapters

Interfaces are the same thing as object-oriented programming [interfaces](#). Adapter refers to a well-known [adapter](#) design pattern that helps separating concerns in object oriented applications.

In *CubicWeb* adapters provide logical functionalities to entity types.

Definition of an adapter is quite trivial. An excerpt from cubicweb itself (found in `cubicweb.entities.adapters`):

```
class ITreeAdapter(EntityAdapter):
    """This adapter has to be overridden to be configured using the
    tree_relation, child_role and parent_role class attributes to
    benefit from this default implementation
    """
    __regid__ = 'ITree'

    child_role = 'subject'
    parent_role = 'object'

    def children_rql(self):
        """returns RQL to get children """
        return self.entity.cw_related_rql(self.tree_relation, self.parent_role)
```

The adapter object has `self.entity` attribute which represents the entity being adapted.

Note: Adapters came with the notion of service identified by the registry identifier of an adapters, hence dropping the need for explicit interface and the `cubicweb.predicates.implements` selector. You should instead use `cubicweb.predicates.is_instance` when you want to select on an entity type, or `cubicweb.predicates.adaptable` when you want to select on a service.

Specializing and binding an adapter

```
from cubicweb.entities.adapters import ITreeAdapter

class MyEntityITreeAdapter(ITreeAdapter):
    __select__ = is_instance('MyEntity')
    tree_relation = 'filed_under'
```

The `ITreeAdapter` here provides a default implementation. The `tree_relation` class attribute is actually used by this implementation to help implement correct behaviour.

Here we provide a specific implementation which will be bound for `MyEntity` entity type (the *adaptee*).

Converting code from Interfaces/Mixins to Adapters

Here we go with a small example. Before:

```
from cubicweb.predicates import implements
from cubicweb.interfaces import ITree
from cubicweb.mixins import ITreeMixIn

class MyEntity(ITreeMixIn, AnyEntity):
    __implements__ = AnyEntity.__implements__ + (ITree,)

class ITreeView(EntityView):
    __select__ = implements('ITree')
    def cell_call(self, row, col):
        entity = self.cw_rset.get_entity(row, col)
        children = entity.children()
```

After:

```
from cubicweb.predicates import adaptable, is_instance
from cubicweb.entities.adapters import ITreeAdapter

class MyEntityITreeAdapter(ITreeAdapter):
    __select__ = is_instance('MyEntity')

class ITreeView(EntityView):
    __select__ = adaptable('ITree')
    def cell_call(self, row, col):
        entity = self.cw_rset.get_entity(row, col)
        itree = entity.cw_adapt_to('ITree')
        children = itree.children()
```

As we can see, the interface/mixin duality disappears and the entity class itself is completely freed from these concerns. When you want to use the ITree interface of an entity, call its `cw_adapt_to` method to get an adapter for this interface, then access to members of the interface on the adapter

Let's look at an example where we defined everything ourselves. We start from:

```
class IFoo(Interface):
    def bar(self, *args):
        raise NotImplementedError

class MyEntity(AnyEntity):
    __regid__ = 'MyEntity'
    __implements__ = AnyEntity.__implements__ + (IFoo,)

    def bar(self, *args):
        return sum(captain.age for captain in self.captains)

class FooView(EntityView):
    __regid__ = 'mycube.fooview'
    __select__ = implements('IFoo')

    def cell_call(self, row, col):
        entity = self.cw_rset.get_entity(row, col)
        self.w('bar: %s' % entity.bar())
```

Converting to:

```
class IFooAdapter(EntityAdapter):
    __regid__ = 'IFoo'
    __select__ = is_instance('MyEntity')

    def bar(self, *args):
        return sum(captain.age for captain in self.entity.captains)

class FooView(EntityView):
    __regid__ = 'mycube.fooview'
    __select__ = adaptable('IFoo')

    def cell_call(self, row, col):
        entity = self.cw_rset.get_entity(row, col)
        self.w('bar: %s' % entity.cw_adapt_to('IFoo').bar())
```

Note: When migrating an entity method to an adapter, the code can be moved as is except for the *self* of the entity class, which in the adapter must become *self.entity*.

Adapters defined in the library

some basic entity adapter implementations, for interfaces used in the framework itself.

class cubicweb.entities.adapters.**CWUserRDFAdapter**(*_cw*, ***kwargs*)

triples()

return sequence of 3-tuple of rdflib identifiers

class cubicweb.entities.adapters.**EntityRDFAdapter**(*_cw*, ***kwargs*)

EntityRDFAdapter is to be specialized for each entity that wants to be converted to RDF using the mechanism from cubicweb.rdf

triples()

return sequence of 3-tuple of rdflib identifiers

uri

<wrapped by the cachedproperty decorator>

class cubicweb.entities.adapters.**IDownloadableAdapter**(*_cw*, ***kwargs*)

interface for downloadable entities

download_content_type()

return MIME type (unicode) of the downloadable content

download_data()

return actual data (bytes) of the downloadable content

download_encoding()

return encoding (unicode) of the downloadable content

download_file_name()

return file name (unicode) of the downloadable content

download_url(***kwargs*)

return a URL to download entity's content

It should be a unicode object containing url-encoded ASCII.

class cubicweb.entities.adapters.**IDublinCoreAdapter**(*_cw*, ***kwargs*)

authors()

Return a suitable description for the author(s) of the entity

creator()

Return a suitable description for the creator of the entity

date(*date_format=None*)

Return latest modification date of entity

description(*format='text/plain'*)

Return a suitable description for entity

language()

Return language used by this entity (translated)

long_title()

Return a more detailed title for entity

title()

Return a suitable *unicode* title for entity

type(form="")

Return the display name for the type of entity (translated)

class cubicweb.entities.adapters.**IEmailableAdapter**(*_cw*, ***kwargs*)

allowed_massmail_keys()

returns a set of allowed email substitution keys

The default is to return the entity's attribute list but you might override this method to allow extra keys. For instance, a Person class might want to return a *companyname* key.

as_email_context()

returns the dictionary as used by the sendmail controller to build email bodies.

NOTE: the dictionary keys should match the list returned by the *allowed_massmail_keys* method.

class cubicweb.entities.adapters.**IFTIndexableAdapter**(*_cw*, ***kwargs*)

standard adapter to handle fulltext indexing

fti_containers(*_done=None*)

return the list of entities to index when handling *self.entity*

The actual list of entities depends on *fulltext_container* usage in the datamodel definition

get_words()

used by the full text indexer to get words to index

this method should only be used on the repository side since it depends on the logilab.database package

Return type list

Returns the list of indexable word of this entity

fti_containers(*_done=None*)

return the list of entities to index when handling *self.entity*

The actual list of entities depends on *fulltext_container* usage in the datamodel definition

get_words()

used by the full text indexer to get words to index

this method should only be used on the repository side since it depends on the logilab.database package

Return type list

Returns the list of indexable word of this entity

class cubicweb.entities.adapters.**INotifiableAdapter**(*_cw*, ***kwargs*)

notification_references(*view*)

used to control References field of email send on notification for this entity. *view* is the notification view.

Should return a list of eids which can be used to generate message identifiers of previously sent email(s)

class cubicweb.entities.adapters.**ISerializableAdapter**(*_cw*, ***kwargs*)

Adapter to serialize an entity to a bare python structure that may be directly serialized to e.g. JSON.

class cubicweb.entities.adapters.ITreeAdapter(_cw, **kwargs)

This adapter provides a tree interface.

It has to be overridden to be configured using the `tree_relation`, `child_role` and `parent_role` class attributes to benefit from this default implementation.

This class provides the following methods:

children(*entities=True*, *sametype=False*)

Return children entities.

According to the *entities* parameter, return entity objects or the equivalent result set.

children_rql()

Returns RQL to get the children of the entity.

different_type_children(*entities=True*)

Return children entities of different type as this entity.

According to the *entities* parameter, return entity objects or the equivalent result set.

is_leaf()

Returns True if the entity does not have any children.

is_root()

Returns true if the entity is root of the tree (e.g. has no parent).

iterchildren(*_done=None*)

Return an iterator over the item's children.

iterparents(*strict=True*)

Return an iterator on the parents of the entity.

parent()

Returns the parent entity if any, else None (e.g. if we are on the root).

path(**kwargs)

Returns the list of eids from the root object to this object.

prefixiter(*_done=None*)

Return an iterator over the item's descendants in a prefixed order.

root()

Return the root entity of the tree.

same_type_children(*entities=True*)

Return children entities of the same type as this entity.

According to the *entities* parameter, return entity objects or the equivalent result set.

class cubicweb.entities.adapters.IUserFriendlyCheckConstraint(*args, **kwargs)

class cubicweb.entities.adapters.IUserFriendlyError(*args, **kwargs)

class cubicweb.entities.adapters.IUserFriendlyUniqueTogether(*args, **kwargs)

More are defined in `web/views`.

5.4.7 How to use entities objects and adapters

The previous chapters detailed the classes and methods available to the developer at the so-called [ORM](#) level. However they say little about the common patterns of usage of these objects.

Entities objects (and their adapters) are used in the repository and web sides of CubicWeb. On the repository side of things, one should manipulate them in Hooks and Operations.

Hooks and Operations provide support for the implementation of rules such as computed attributes, coherency invariants, etc (they play the same role as database triggers, but in a way that is independent of the actual data sources).

So a lot of an application's business rules will be written in Hooks (or Operations).

On the web side, views also typically operate using entity objects. Obvious entity methods for use in views are the Dublin Core methods like `dc_title`. For separation of concerns reasons, one should ensure no ui logic pervades the entities level, and also no business logic should creep into the views.

In the duration of a transaction, entities objects can be instantiated many times, in views and hooks, even for the same database entity. For instance, in a classic CubicWeb deployment setup, the repository and the web front-end are separated process communicating over the wire. There is no way state can be shared between these processes (there is a specific API for that). Hence, it is not possible to use entity objects as messengers between these components of an application. It means that an attribute set as in `obj.x = 42`, whether or not `x` is actually an entity schema attribute, has a short life span, limited to the hook, operation or view within which the object was built.

Setting an attribute or relation value can be done in the context of a Hook/Operation, using the `obj.cw_set(x=42)` notation or a plain RQL SET expression.

In views, it would be preferable to encapsulate the necessary logic in a method of an adapter for the concerned entity class(es). But of course, this advice is also reasonable for Hooks/Operations, though the separation of concerns here is less stringent than in the case of views.

This leads to the practical role of objects adapters: it's where an important part of the application logic lies (the other part being located in the Hook/Operations).

5.4.8 Anatomy of an entity class

We can look now at a real life example coming from the [tracker](#) cube. Let us begin to study the `entities/project.py` content.

```
from cubicweb.entities.adapters import ITreeAdapter

class ProjectAdapter(ITreeAdapter):
    __select__ = is_instance('Project')
    tree_relation = 'subproject_of'

class Project(AnyEntity):
    __regid__ = 'Project'
    fetch_attrs, cw_fetch_order = fetch_config(('name', 'description',
                                                'description_format', 'summary'))

    TICKET_DEFAULT_STATE_RESTRICTION = 'S name IN ("created","identified","released","scheduled"
↪)"'

    def dc_title(self):
        return self.name
```


The fact that the *Project* entity type implements an *ITree* interface is materialized by the *ProjectAdapter* class (inheriting the pre-defined *ITreeAdapter* whose `__regid__` is of course *ITree*), which will be selected on *Project* entity types because of its selector. On this adapter, we redefine the `tree_relation` attribute of the *ITreeAdapter* class.

This is typically used in views concerned with the representation of tree-like structures (CubicWeb provides several such views).

It is important that the views themselves try not to implement this logic, not only because such views would be hardly applicable to other tree-like relations, but also because it is perfectly fine and useful to use such an interface in Hooks.

In fact, Tree nature is a property of the data model that cannot be fully and portably expressed at the level of database entities (think about the transitive closure of the child relation). This is a further argument to implement it at entity class level.

`fetch_attrs` configures which attributes should be pre-fetched when using ORM methods retrieving entity of this type. In a same manner, the `cw_fetch_order` is a class method allowing to control sort order. More on this in [Loaded attributes and default sorting management](#).

We can observe the big `TICKET_DEFAULT_STATE_RESTRICTION` is a pure application domain piece of data. There is, of course, no limitation to the amount of class attributes of this kind.

The `dc_title` method provides a (unicode string) value likely to be consumed by views, but note that here we do not care about output encodings. We care about providing data in the most universal format possible, because the data could be used by a web view (which would be responsible of ensuring XHTML compliance), or a console or file oriented output (which would have the necessary context about the needed byte stream encoding).

Note: The Dublin Core `dc_XXX` methods are not moved to an adapter as they are extremely prevalent in CubicWeb and assorted cubes and should be available for all entity types.

Let us now dig into more substantial pieces of code, continuing the *Project* class.

```
def latest_version(self, states=('published',), reverse=None):
    """returns the latest version(s) for the project in one of the given
    states.

    when no states specified, returns the latest published version.
    """
    order = 'DESC'
    if reverse is not None:
        warn('reverse argument is deprecated',
             DeprecationWarning, stacklevel=1)
        if reverse:
            order = 'ASC'
    rset = self.versions_in_state(states, order, True)
    if rset:
        return rset.get_entity(0, 0)
    return None

def versions_in_state(self, states, order='ASC', limit=False):
    """returns version(s) for the project in one of the given states, sorted
    by version number.

    If limit is true, limit result to one version.
    If reverse, versions are returned from the smallest to the greatest.
```

(continues on next page)

(continued from previous page)

```
"""
if limit:
    order += ' LIMIT 1'
rql = 'Any V,N ORDERBY version_sort_value(N) %s ' \
      'WHERE V num N, V in_state S, S name IN (%s), ' \
      'V version_of P, P eid %(p)s' % (order, ','.join(repr(s) for s in states))
return self._cw.execute(rql, {'p': self.eid})
```

These few lines exhibit the important properties we want to outline:

- entity code is concerned with the application domain
- it is NOT concerned with database consistency (this is the realm of Hooks/Operations); in other words, it assumes a consistent world
- it is NOT (directly) concerned with end-user interfaces
- however it can be used in both contexts
- it does not create or manipulate the internal object's state
- it plays freely with RQL expression as needed
- it is not concerned with internationalization
- it does not raise exceptions

5.5 Core APIs

5.5.1 Request and ResultSet methods

Those are methods you'll find on both request objects and on repository session.

Request methods

URL handling:

- *build_url(*args, **kwargs)*, returns an absolute URL based on the given arguments. The *controller* supposed to handle the response, can be specified through the first positional parameter (the connection is theoretically done automatically :).

Data formatting:

- *format_date(date, date_format=None, time=False)* returns a string for a date time according to instance's configuration
- *format_time(time)* returns a string for a date time according to instance's configuration

And more...:

- *tal_render(template, variables)*, renders a precompiled page template with variables in the given dictionary as context

Result set methods

- `get_entity(row, col)`, returns the entity corresponding to the data position in the *result set*
- `complete_entity(row, col, skip_bytes=True)`, is equivalent to `get_entity` but also call the method `complete()` on the entity before returning it

5.6 Repository customization

5.6.1 Sessions

Sessions are objects linked to an authenticated user. The `Session.new_cnx` method returns a new `Connection` linked to that session.

5.6.2 Connections

Connections provide the `.execute` method to query the data sources, along with `.commit` and `.rollback` methods for transaction management.

Kinds of connections

There are two kinds of connections.

- *normal connections* are the most common: they are related to users and carry security checks coming with user credentials
- *internal connections* have all the powers; they are also used in only a few situations where you don't already have an adequate session at hand, like: user authentication, data synchronisation in multi-source contexts

Normal connections are typically named `_cw` in most appobjects or sometimes just *session*.

Internal connections are available from the *Repository* object and are to be used like this:

```
with self.repo.internal_cnx() as cnx:
    do_stuff_with(cnx)
    cnx.commit()
```

Connections should always be used as context managers, to avoid leaks.

Python/RQL API

The Python API developped to interface with RQL is inspired from the standard db-api, but since `execute` returns its results directly, there is no *cursor* concept.

```
execute(rqlstring, args=None, build_descr=True)
```

rqlstring the RQL query to execute (unicode)

args if the query contains substitutions, a dictionary containing the values to use

The *Connection* object owns the methods `commit` and `rollback`. You *should never need to use them* during the development of the web interface based on the *CubicWeb* framework as it determines the end of the transaction depending on the query execution success. They are however useful in other contexts such as tests or custom controllers.

Note: If a query generates an error related to security (`Unauthorized`) or to integrity (`ValidationError`), the transaction can still continue but you won't be able to commit it, a rollback will be necessary to start a new transaction.

Also, a rollback is automatically done if an error occurs during commit.

Note: A `ValidationError` has a `entity` attribute. In CubicWeb, this attribute is set to the entity's `eid` (not a reference to the entity itself).

Executing RQL queries from a view or a hook

When you're within code of the web interface, the `Connection` is handled by the request object. You should not have to access it directly, but use the `execute` method directly available on the request, eg:

```
rset = self._cw.execute(rqlstring, kwargs)
```

Similarly, on the server side (eg in hooks), there is no request object (since you're directly inside the data-server), so you'll have to use the `execute` method of the `Connection` object.

Proper usage of `.execute`

Let's say you want to get `T` which is in configuration `C`, this translates to:

```
self._cw.execute('Any T WHERE T in_conf C, C eid %s' % entity.eid)
```

But it must be written in a syntax that will benefit from the use of a cache on the RQL server side:

```
self._cw.execute('Any T WHERE T in_conf C, C eid %(x)s', {'x': entity.eid})
```

The syntax tree is built once for the “generic” RQL and can be re-used with a number of different eids. The `rql IN` operator is an exception to this rule.

```
self._cw.execute('Any T WHERE T in_conf C, C name IN (%s)'
                 % ','.join(['foo', 'bar']))
```

Alternatively, some of the common data related to an entity can be obtained from the `entity.related()` method (which is used under the hood by the ORM when you use attribute access notation on an entity to get a relation. The initial request would then be translated to:

```
entity.related('in_conf', 'object')
```

Additionally this benefits from the `fetch_attrs` policy (see *Loaded attributes and default sorting management*) optionally defined on the class element, which says which attributes must be also loaded when the entity is loaded through the ORM.

The *ResultSet* API

ResultSet instances are a very commonly manipulated object. They have a rich API as seen below, but we would like to highlight a bunch of methods that are quite useful in day-to-day practice:

- `__str__()` (applied by `print`) gives a very useful overview of both the underlying RQL expression and the data inside; unavoidable for debugging purposes
- `printable_rql()` returns a well formed RQL expression as a string; it is very useful to build views
- `entities()` returns a generator on all entities of the result set
- `get_entity(row, col)` gets the entity at row, col coordinates; one of the most used result set methods

class `cubicweb.rset.ResultSet`(*results, rql, args=None, description=None, variables=None*)

A result set wraps a RQL query result. This object implements partially the list protocol to allow direct use as a list of result rows.

Parameters

- **rowcount** (*int*) – number of rows in the result
- **rows** (*list*) – list of rows of result
- **description** (*list*) – result’s description, using the same structure as the result itself
- **rql** (*str or unicode*) – the original RQL query string

all(*col=0*)

iter on entities with eid in the *col* column of the result set

column_types(***kwargs*)

return the list of different types in the column with the given col

Parameters **col** (*int*) – the index of the desired column

Return type *list*

Returns the different entities type found in the column

complete_entity(*row, col=0, skip_bytes=True*)

short cut to get an completed entity instance for a particular row (all instance’s attributes have been fetched)

description_struct(***kwargs*)

return a list describing sequence of results with the same description, e.g. : `[[0, 4, ('Bug',)] [[0, 4, ('Bug',), [5, 8, ('Story',)] [[0, 3, ('Project', 'Version',)]]`

entities(*col=0*)

iter on entities with eid in the *col* column of the result set

filtered_rset(*filtercb, col=0*)

filter the result set according to a given filtercb

Parameters

- **filtercb** (*callable(entity)*) – a callable which should take an entity as argument and return False if it should be skipped, else True
- **col** (*int*) – the column index

Return type *ResultSet*

first(*col=0*)

Retrieve the first entity from the query.

If the result set is empty, raises `NoResultError`.

Parameters **col** (*int*) – The column localising the entity in the unique row

Returns the partially initialized *Entity* instance

get_entity(***kwargs*)

convenience method for query retrieving a single entity, returns a partially initialized Entity instance.

Warning: Due to the cache wrapping this function, you should NEVER give row as a named parameter (i.e. `rset.get_entity(0, 1)` is OK but `rset.get_entity(row=0, col=1)` isn't)

Parameters **row, col** (*int*, *int*) – row and col numbers localizing the entity among the result's table

Returns the partially initialized *Entity* instance

iter_rows_with_entities()

iterates over rows, and for each row eids are converted to plain entities

last(*col=0*)

Retrieve the last entity from the query.

If the result set is empty, raises `NoResultError`.

Parameters **col** (*int*) – The column localising the entity in the unique row

Returns the partially initialized *Entity* instance

limit(*limit, offset=0, inplace=False*)

limit the result set to the given number of rows optionally starting from an index different than 0

Parameters

- **limit** (*int*) – the maximum number of results
- **offset** (*int*) – the offset index
- **inplace** (*bool*) – if true, the result set is modified in place, else a new result set is returned and the original is left unmodified

Return type *ResultSet*

limited_rql()

returns a printable rql for the result set associated to the object, with limit/offset correctly set according to maximum page size and currently displayed page when necessary

one(*col=0*)

Retrieve exactly one entity from the query.

If the result set is empty, raises `NoResultError`. If the result set has more than one row, raises `MultipleResultsError`.

Parameters **col** (*int*) – The column localising the entity in the unique row

Returns the partially initialized *Entity* instance

possible_actions(kwargs)**

Return possible actions on this result set. Should always be called with the same arguments so it may be computed only once.

printable_rql()

return the result set's origin rql as a string, with arguments substituted

related_entity(kwargs)**

given an cell of the result set, try to return a (entity, relation name) tuple to which this cell is linked.

This is especially useful when the cell is an attribute of an entity, to get the entity to which this attribute belongs to.

searched_text(kwargs)**

returns the searched text in case of full-text search

Returns searched text or *None* if the query is not a full-text query

sorted_rset(keyfunc, reverse=False, col=0)

sorts the result set according to a given keyfunc

Parameters

- **keyfunc** (*callable(entity)*) – a callable which should take an entity as argument and return the value used to compare and sort
- **reverse** (*bool*) – if the result should be reversed
- **col** (*int*) – the column index. if col = -1, the whole row are used

Return type *ResultSet*

split_rset(keyfunc=None, col=0, return_dict=False)

splits the result set in multiple result sets according to a given key

Parameters

- **keyfunc** (*callable(entity or FinalType)*) – a callable which should take a value of the rset in argument and return the value used to group the value. If not define, raw value of the specified columns is used.
- **col** (*int*) – the column index. if col = -1, the whole row are used
- **return_dict** (*Boolean*) – If true, the function return a mapping (key -> rset) instead of a list of rset

Return type List of *ResultSet* or mapping of *ResultSet*

syntax_tree(kwargs)**

Return the **cached** syntax tree (`rql.stmts.Union`) for the originating query.

You can expect it to have solutions computed and it will be properly annotated. Since this is a cached shared object, **you must not modify it**.

transformed_rset(transformcb)

the result set according to a given column types

Parameters

- **transformcb** – a callable which should take a row and its type description as parameters, and return the transformed row and type description.
- **col** (*int*) – the column index

Return type *ResultSet*

Authentication and management of sessions

The authentication process is a ballet involving a few dancers:

- through its *get_session* method the top-level application object (the *CubicWebPublisher*) will open a session whenever a web request comes in; it asks the *session manager* to open a session (giving the web request object as context) using *open_session*
 - the session manager asks its authentication manager (which is a *component*) to authenticate the request (using *authenticate*)
 - * the authentication manager asks, in order, to its authentication information retrievers, a login and an opaque object containing other credentials elements (calling *authentication_information*), giving the request object each time
 - the default retriever (named *LoginPasswordRetriever*) will in turn defer login and password fetching to the request object (which, depending on the authentication mode (*cookie* or *http*), will do the appropriate things and return a login and a password)
 - * the authentication manager, on success, asks the *Repository* object to connect with the found credentials (using *connect*)
 - the repository object asks authentication to all of its sources which support the *CWUser* entity with the given credentials; when successful it can build the *cwuser* entity, from which a regular *Session* object is made; it returns the session id
 - the source in turn will delegate work to an authenticifier class that defines the ultimate *authenticate* method (for instance the native source will query the database against the provided credentials)
 - * the authentication manager, on success, will call back *_all_* retrievers with *authenticated* and return its authentication data (on failure, it will try the anonymous login or, if the configuration forbids it, raise an *AuthenticationError*)

Writing authentication plugins

Sometimes CubicWeb's out-of-the-box authentication schemes (cookie and http) are not sufficient. Nowadays there is a plethora of such schemes and the framework cannot provide them all, but as the sequence above shows, it is extensible.

Two levels have to be considered when writing an authentication plugin: the web client and the repository.

We invented a scenario where it makes sense to have a new plugin in each side: some middleware will do pre-authentication and under the right circumstances add a new HTTP *x-foo-user* header to the query before it reaches the CubicWeb instance. For a concrete example of this, see the [trustedauth](#) cube.

Repository authentication plugins

On the repository side, it is possible to register a source authenticifier using the following kind of code:

```
from cubicweb.server.sources import native

class FooAuthenticator(native.LoginPasswordAuthenticator):
    """ a source authenticifier plugin
    if 'foo' in authentication information, no need to check
    password
```

(continues on next page)

(continued from previous page)

```

"""
auth_rql = 'Any X WHERE X is CWUser, X login %(login)s'

def authenticate(self, session, login, **kwargs):
    """return CWUser eid for the given login
    if this account is defined in this source,
    else raise `AuthenticationError`
    """
    session.debug('authentication by %s', self.__class__.__name__)
    if 'foo' not in kwargs:
        return super(FooAuthentifier, self).authenticate(session, login, **kwargs)
    try:
        rset = session.execute(self.auth_rql, {'login': login})
        return rset[0][0]
    except Exception, exc:
        session.debug('authentication failure (%s)', exc)
        raise AuthenticationError('foo user is unknown to us')

```

Since repository authenticifiers are not appobjects, we have to register them through a *server_startup* hook.

```

class ServerStartupHook(hook.Hook):
    """ register the foo authenticator """
    __regid__ = 'fooauthenticatorregisterer'
    events = ('server_startup',)

    def __call__(self):
        self.debug('registering foo authenticifier')
        self.repo.system_source.add_authentifier(FooAuthentifier())

```

Web authentication plugins

```

class XFooUserRetriever(authentication.LoginPasswordRetriever):
    """ authenticate by the x-foo-user http header
    or just do normal login/password authentication
    """
    __regid__ = 'x-foo-user'
    order = 0

    def authentication_information(self, req):
        """retrieve authentication information from the given request, raise
        NoAuthInfo if expected information is not found
        """
        self.debug('web authenticator building auth info')
        try:
            login = req.get_header('x-foo-user')
            if login:
                return login, {'foo': True}
            else:
                return super(XFooUserRetriever, self).authentication_information(self,
↪req)

```

(continues on next page)

(continued from previous page)

```

except Exception, exc:
    self.debug('web authenticator failed (%s)', exc)
    raise authentication.NoAuthInfo()

def authenticated(self, retriever, req, cnx, login, authinfo):
    """callback when return authentication information have opened a
    repository connection successfully. Take care req has no session
    attached yet, hence req.execute isn't available.

    Here we set a flag on the request to indicate that the user is
    foo-authenticated. Can be used by a selector
    """
    self.debug('web authenticator running post authentication callback')
    cnx.foo_user = authinfo.get('foo')

```

In the *authenticated* method we add (in an admittedly slightly hackish way) an attribute to the connection object. This, in turn, can be used to build a selector dispatching on the fact that the user was preauthenticated or not.

```

@objectify_selector
def foo_authenticated(cls, req, rset=None, **kwargs):
    if hasattr(req.cnx, 'foo_user') and req.foo_user:
        return 1
    return 0

```

Full Session and Connection API

class cubicweb.server.session.**Connection**(*repo, user*)

Repository Connection

Holds all connection related data

Database connection resources:

hooks_in_progress, boolean flag telling if the executing query is coming from a repoapi connection or is a query from within the repository (e.g. started by hooks)

cnxset, the connections set to use to execute queries on sources. If the transaction is read only, the connection set may be freed between actual queries. This allows multiple connections with a reasonably low connection set pool size.

mode, string telling the connections set handling mode, may be one of 'read' (connections set may be freed), 'write' (some write was done in the connections set, it can't be freed before end of the transaction), 'transaction' (we want to keep the connections set during all the transaction, with or without writing)

Shared data:

data is a dictionary bound to the underlying session, who will be present for the life time of the session. This may be useful for web clients that rely on the server for managing bits of session-scoped data.

transaction_data is a dictionary cleared at the end of the transaction. Hooks and operations may put arbitrary data in there.

Internal state:

`pending_operations`, ordered list of operations to be processed on commit/rollback

`commit_state`, describing the transaction commit state, may be one of `None` (not yet committing), `'precommit'` (calling precommit event on operations), `'postcommit'` (calling postcommit event on operations), `'uncommitable'` (some `ValidationError` or `Unauthorized` error has been raised during the transaction and so it must be rolled back).

Hooks controls:

`deny_all_hooks_but(*categories)`

Context manager to disable all hooks but those in the given categories.

`allow_all_hooks_but(*categories)`

Context manager to enable all hooks but those in the given categories.

Security level Management:

`read_security` and `write_security`, boolean flags telling if read/write security is currently activated.

5.6.3 Hooks and Operations

Generalities

Paraphrasing the [emacs](#) documentation, let us say that hooks are an important mechanism for customizing an application. A hook is basically a list of functions to be called on some well-defined occasion (this is called *running the hook*).

Hooks

In *CubicWeb*, hooks are subclasses of the [Hook](#) class. They are selected over a set of pre-defined *events* (and possibly more conditions, hooks being selectable appobjects like views and components). They should implement a `__call__()` method that will be called when the hook is triggered.

There are two families of events: data events (before / after any individual update of an entity / or a relation in the repository) and server events (such as server startup or shutdown). In a typical application, most of the hooks are defined over data events.

Also, some [Operation](#) may be registered by hooks, which will be fired when the transaction is committed or rolled back.

The purpose of data event hooks is usually to complement the data model as defined in the schema, which is static by nature and only provide a restricted builtin set of dynamic constraints, with dynamic or value driven behaviours. For instance they can serve the following purposes:

- enforcing constraints that the static schema cannot express (spanning several entities/relations, exotic value ranges and cardinalities, etc.)
- implement computed attributes

It is functionally equivalent to a [database trigger](#), except that database triggers definition languages are not standardized, hence not portable (for instance, PL/SQL works with Oracle and PostgreSQL but not SQLite).

Hint: It is a good practice to write unit tests for each hook. See an example in [Unit test by example](#)

Operations

Operations are subclasses of the *Operation* class that may be created by hooks and scheduled to happen on *precommit*, *postcommit* or *rollback* event (i.e. respectively before/after a commit or before a rollback of a transaction).

Hooks are being fired immediately on data operations, and it is sometime necessary to delay the actual work down to a time where we can expect all information to be there, or when all other hooks have run (though take care since operations may themselves trigger hooks). Also while the order of execution of hooks is data dependant (and thus hard to predict), it is possible to force an order on operations.

So, for such case where you may miss some information that may be set later in the transaction, you should instantiate an operation in the hook.

Operations may be used to:

- implements a validation check which needs that all relations be already set on an entity
- process various side effects associated with a transaction such as filesystem updates, mail notifications, etc.

Events

Hooks are mostly defined and used to handle *dataflow* operations. It means as data gets in (entities added, updated, relations set or unset), specific events are issued and the Hooks matching these events are called.

You can get the event that triggered a hook by accessing its *event* attribute.

Entity modification related events

When called for one of these events, hook will have an *entity* attribute containing the entity instance.

- *before_add_entity*, *before_update_entity*:

On those events, you can access the modified attributes of the entity using the *entity.cw_edited* dictionary. The values can be modified and the old values can be retrieved.

If you modify the *entity.cw_edited* dictionary in the hook, that is before the database operations take place, you will avoid the need to process a whole new rql query and the underlying backend query (eg usually sql) will contain the modified data. For example:

```
self.entity.cw_edited['age'] = 42
```

will modify the age before it is written to the backend storage.

Similarly, removing an attribute from *cw_edited* will cancel its modification:

```
del self.entity.cw_edited['age']
```

On a *before_update_entity* event, you can access the old and new values:

```
old, new = entity.cw_edited.oldnewvalue('age')
```

- *after_add_entity*, *after_update_entity*

On those events, you can get the list of attributes that were modified using the *entity.cw_edited* dictionary, but you can not modify it or get the old value of an attribute.

- *before_delete_entity*, *after_delete_entity*

On those events, the entity has no *cw_edited* dictionary.

Note: `self.entity.cw_set(age=42)` will set the `age` attribute to 42. But to do so, it will generate a rql query that will have to be processed, hence may trigger some hooks, etc. This could lead to infinitely looping hooks.

Relation modification related events

When called for one of these events, hook will have `eidfrom`, `rtype`, `eidto` attributes containing respectively the eid of the subject entity, the relation type and the eid of the object entity.

- `before_add_relation`, `before_delete_relation`

On those events, you can still get the original relation by issuing a rql query.

- `after_add_relation`, `after_delete_relation`

Specific selectors are shipped for these kinds of events, see in particular `match_rtype`.

Also note that relations can be added or deleted, but not updated.

Non data events

Hooks called on server start/maintenance/stop event (e.g. `server_startup`, `server_maintenance`, `before_server_shutdown`, `server_shutdown`) have a `repo` attribute, but their ``_cw`` attribute is `None`. The `server_startup` is called on regular startup, while `server_maintenance` is called on cubicweb-ctl upgrade or shell commands. `server_shutdown` is called anyway but connections to the native source is impossible; `before_server_shutdown` handles that.

Hooks called on backup/restore event (eg `server_backup`, `server_restore`) have a `repo` and a `timestamp` attributes, but their ``_cw`` attribute is `None`.

API

Hooks control

It is sometimes convenient to explicitly enable or disable some hooks. For instance if you want to disable some integrity checking hook. This can be controlled more finely through the `category` class attribute, which is a string giving a category name. One can then uses the `deny_all_hooks_but()` and `allow_all_hooks_but()` context managers to explicitly enable or disable some categories.

The existing categories are:

- `security`, security checking hooks
- `workflow`, workflow handling hooks
- `metadata`, hooks setting meta-data on newly created entities
- `notification`, email notification hooks
- `integrity`, data integrity checking hooks
- `activeintegrity`, data integrity consistency hooks, that you should **never** want to disable
- `syncsession`, hooks synchronizing existing sessions
- `syncschema`, hooks synchronizing instance schema (including the physical database)
- `email`, email address handling hooks

- bookmark, bookmark entities handling hooks

Nothing precludes one to invent new categories and use existing mechanisms to filter them in or out.

Hooks specific predicates

class cubicweb.server.hook.**match_rtype**(*expected, **more)

accept if the relation type is found in expected ones. Optional named parameters *fromtypes* and *toetypes* can be used to restrict target subject and/or object entity types of the relation.

Parameters

- ***expected** – possible relation types
- **fromtypes** – candidate entity types as subject of relation
- **toetypes** – candidate entity types as object of relation

class cubicweb.server.hook.**match_rtype_sets**(*expected)

accept if the relation type is in one of the sets given as initializer argument. The goal of this predicate is that it keeps reference to original sets, so modification to those sets are considered by the predicate. For instance

```
MYSET = set()

class Hook1(Hook):
    __regid__ = 'hook1'
    __select__ = Hook.__select__ & match_rtype_sets(MYSET)
    ...

class Hook2(Hook):
    __regid__ = 'hook2'
    __select__ = Hook.__select__ & match_rtype_sets(MYSET)
```

Client code can now change *MYSET*, this will change the selection criteria of Hook1 and Hook2.

Hooks and operations classes

class cubicweb.server.hook.**Hook**(req, event, **kwargs)

Base class for hook.

Hooks being appobjects like views, they have a `__regid__` and a `__select__` class attribute. Like all appobjects, hooks have the `self._cw` attribute which represents the current connection. In entity hooks, a `self.entity` attribute is also present.

The *events* tuple is used by the base class selector to dispatch the hook on the right events. It is possible to dispatch on multiple events at once if needed (though take care as hook attribute may vary as described above).

Note: Do not forget to extend the base class selectors as in:

```
class MyHook(Hook):
    __regid__ = 'whatever'
    __select__ = Hook.__select__ & is_instance('Person')
```

else your hooks will be called madly, whatever the event.

class cubicweb.server.hook.Operation(*cnx*, ***kwargs*)

Base class for operations.

Operation may be instantiated in the hooks' `__call__` method. It always takes a connection object as first argument (accessible as `.cnx` from the operation instance), and optionally all keyword arguments needed by the operation. These keyword arguments will be accessible as attributes from the operation instance.

An operation is triggered on connections set events related to commit / rollback transactions. Possible events are:

- *precommit*:

the transaction is being prepared for commit. You can freely do any heavy computation, raise an exception if the commit can't go. or even add some new operations during this phase. If you do anything which has to be reverted if the commit fails afterwards (eg altering the file system for instance), you'll have to support the 'revertprecommit' event to revert things by yourself

- *revertprecommit*:

if an operation failed while being pre-committed, this event is triggered for all operations which had their 'precommit' event already fired to let them revert things (including the operation which made the commit fail)

- *rollback*:

the transaction has been either rolled back either:

- intentionally
- a 'precommit' event failed, in which case all operations are rolled back once 'revertprecommit' has been called

- *postcommit*:

the transaction is over. All the ORM entities accessed by the earlier transaction are invalid. If you need to work on the database, you need to start a new transaction, for instance using a new internal connection, which you will need to commit.

For an operation to support an event, one has to implement the `<event name>_event` method with no arguments.

The order of operations may be important, and is controlled according to the `insert_index`'s method output (whose implementation vary according to the base hook class used).

class cubicweb.server.hook.LateOperation(*cnx*, ***kwargs*)

special operation which should be called after all possible (ie non late) operations

class cubicweb.server.hook.DataOperationMixin(**args*, ***kwargs*)

Mix-in class to ease applying a single operation on a set of data, avoiding creating as many operations as there are individual modifications. The body of the operation must then iterate over the values that have been stored in a single operation instance.

You should try to use this instead of creating an operation for each *value*, since handling operations becomes costly on massive data import.

Usage looks like:

```
class MyEntityHook(Hook):
    __regid__ = 'my.entity.hook'
    __select__ = Hook.__select__ & is_instance('MyEntity')
    events = ('after_add_entity',)

    def __call__(self):
        MyOperation.get_instance(self._cw).add_data(self.entity)
```

(continues on next page)

(continued from previous page)

```
class MyOperation(DataOperationMixIn, Operation):
    def precommit_event(self):
        for bucket in self.get_data():
            process(bucket)
```

You can modify the *containercls* class attribute, which defines the container class that should be instantiated to hold payloads. An instance is created on instantiation, and then the *add_data()* method will add the given data to the existing container. Default to a *set*. Give *list* if you want to keep arrival ordering. You can also use another kind of container by redefining *_build_container()* and *add_data()*

More optional parameters can be given to the *get_instance* operation, that will be given to the operation constructor (for obvious reasons those parameters should not vary accross different calls to this method for a given operation).

Note: For sanity reason *get_data* will reset the operation, so that once the operation has started its treatment, if some hook want to push additional data to this same operation, a new instance will be created (else that data has a great chance to be never treated). This implies:

- you should **always** call *get_data* when starting treatment
- you should **never** call *get_data* for another reason.

Example using dataflow hooks

We will use a very simple example to show hooks usage. Let us start with the following schema.

```
class Person(EntityType):
    age = Int(required=True)
```

We would like to add a range constraint over a person's age. Let's write an hook (supposing yams can not handle this natively, which is wrong). It shall be placed into *mycube/hooks.py*. If this file were to grow too much, we can easily have a *mycube/hooks/...* package containing hooks in various modules.

```
from cubicweb import ValidationError
from cubicweb.predicates import is_instance
from cubicweb.server.hook import Hook

class PersonAgeRange(Hook):
    __regid__ = 'person_age_range'
    __select__ = Hook.__select__ & is_instance('Person')
    events = ('before_add_entity', 'before_update_entity')

    def __call__(self):
        if 'age' in self.entity.cw_edited:
            if 0 <= self.entity.age <= 120:
                return
            msg = self._cw._('age must be between 0 and 120')
            raise ValidationError(self.entity.eid, {'age': msg})
```

In our example the base *__select__* is augmented with an *is_instance* selector matching the desired entity type.

The *events* tuple is used to specify that our hook should be called before the entity is added or updated.

Then in the hook's `__call__` method, we:

- check if the 'age' attribute is edited
- if so, check the value is in the range
- if not, raise a validation error properly

Now let's augment our schema with a new *Company* entity type with some relation to *Person* (in 'mycube/schema.py').

```
class Company(EntityType):
    name = String(required=True)
    boss = SubjectRelation('Person', cardinality='1*')
    subsidiary_of = SubjectRelation('Company', cardinality='*?')
```

We would like to constrain the company's bosses to have a minimum (legal) age. Let's write an hook for this, which will be fired when the *boss* relation is established (still supposing we could not specify that kind of thing in the schema).

```
class CompanyBossLegalAge(Hook):
    __regid__ = 'company_boss_legal_age'
    __select__ = Hook.__select__ & match_rtype('boss')
    events = ('before_add_relation',)

    def __call__(self):
        boss = self._cw.entity_from_eid(self.eidto)
        if boss.age < 18:
            msg = self._cw._('the minimum age for a boss is 18')
            raise ValidationError(self.eidfrom, {'boss': msg})
```

Note: We use the `match_rtype` selector to select the proper relation type.

The essential difference with respect to an entity hook is that there is no `self.entity`, but `self.eidfrom` and `self.eidto` hook attributes which represent the subject and object **eid** of the relation.

Suppose we want to check that there is no cycle by the *subsidiary_of* relation. This is best achieved in an operation since all relations are likely to be set at commit time.

```
from cubicweb.server.hook import Hook, DataOperationMixIn, Operation, match_rtype

def check_cycle(session, eid, rtype, role='subject'):
    parents = set([eid])
    parent = session.entity_from_eid(eid)
    while parent.related(rtype, role):
        parent = parent.related(rtype, role)[0]
        if parent.eid in parents:
            msg = session._('detected %s cycle' % rtype)
            raise ValidationError(eid, {rtype: msg})
        parents.add(parent.eid)

class CheckSubsidiaryCycleOp(Operation):

    def precommit_event(self):
```

(continues on next page)

(continued from previous page)

```

        check_cycle(self.session, self.eidto, 'subsidiary_of')

class CheckSubsidiaryCycleHook(Hook):
    __regid__ = 'check_no_subsidiary_cycle'
    __select__ = Hook.__select__ & match_rtype('subsidiary_of')
    events = ('after_add_relation',)

    def __call__(self):
        CheckSubsidiaryCycleOp(self._cw, eidto=self.eidto)

```

Like in hooks, *ValidationError* can be raised in operations. Other exceptions are usually programming errors.

In the above example, our hook will instantiate an operation each time the hook is called, i.e. each time the *subsidiary_of* relation is set. There is an alternative method to schedule an operation from a hook, using the *get_instance()* class method.

```

class CheckSubsidiaryCycleHook(Hook):
    __regid__ = 'check_no_subsidiary_cycle'
    events = ('after_add_relation',)
    __select__ = Hook.__select__ & match_rtype('subsidiary_of')

    def __call__(self):
        CheckSubsidiaryCycleOp.get_instance(self._cw).add_data(self.eidto)

class CheckSubsidiaryCycleOp(DataOperationMixIn, Operation):

    def precommit_event(self):
        for eid in self.get_data():
            check_cycle(self.session, eid, self.rtype)

```

Here, we call *add_data()* so that we will simply accumulate eids of entities to check at the end in a single *CheckSubsidiaryCycleOp* operation. Values are stored in a set associated to the ‘check_no_subsidiary_cycle’ transaction data key. The set initialization and operation creation are handled nicely by *add_data()*.

A more realistic example can be found in the advanced tutorial chapter *Step 2: security propagation in hooks*.

Inter-instance communication

If your application consists of several instances, you may need some means to communicate between them. Cubicweb provides a publish/subscribe mechanism using ØMQ. In order to use it, use *add_subscription()* on the *repo.app_instances_bus* object. The *callback* will get the message (as a list). A message can be sent by calling *publish()* on *repo.app_instances_bus*. The first element of the message is the topic which is used for filtering and dispatching messages.

```

class FooHook(hook.Hook):
    events = ('server_startup',)
    __regid__ = 'foo_startup'

    def __call__(self):
        def callback(msg):
            self.info('received message: %s', ' '.join(msg))
            self.repo.app_instances_bus.add_subscription('hello', callback)

```

```
def do_foo(self):
    actually_do_foo()
    self._cw.repo.app_instances_bus.publish(['hello', 'world'])
```

The *zmq-address-pub* configuration variable contains the address used by the instance for sending messages, e.g. *tcp://*:1234*. The *zmq-address-sub* variable contains a comma-separated list of addresses to listen on, e.g. *tcp://localhost:1234, tcp://192.168.1.1:2345*.

Hooks writing tips

Reminder

You should never use the *entity.foo = 42* notation to update an entity. It will not do what you expect (updating the database). Instead, use the *cw_set()* method or direct access to entity's *cw_edited* attribute if you're writing a hook for 'before_add_entity' or 'before_update_entity' event.

How to choose between a before and an after event ?

*before_** hooks give you access to the old attribute (or relation) values. You can also intercept and update edited values in the case of entity modification before they reach the database.

Else the question is: should I need to do things before or after the actual modification ? If the answer is "it doesn't matter", use an 'after' event.

Validation Errors

When a hook which is responsible to maintain the consistency of the data model detects an error, it must use a specific exception named *ValidationError*. Raising anything but a (subclass of) *ValidationError* is a programming error. Raising it entails aborting the current transaction.

This exception is used to convey enough information up to the user interface. Hence its constructor is different from the default Exception constructor. It accepts, positionally:

- an entity eid (**not the entity itself**),
- a dict whose keys represent attribute (or relation) names and values an end-user facing message (hence properly translated) relating the problem.

```
raise ValidationError(earth.eid, {'sea_level': self._cw._('too high'),
                                   'temperature': self._cw._('too hot')})
```

Checking for object created/deleted in the current transaction

In hooks, you can use the *added_in_transaction()* or *deleted_in_transaction()* of the session object to check if an eid has been created or deleted during the hook's transaction.

This is useful to enable or disable some stuff if some entity is being added or deleted.

```
if self._cw.deleted_in_transaction(self.eidto):
    return
```

Peculiarities of inlined relations

Relations which are defined in the schema as *inlined* (see *Relation type* for details) are inserted in the database at the same time as entity attributes.

This may have some side effect, for instance when creating an entity and setting an inlined relation in the same rql query, then at *before_add_relation* time, the relation will already exist in the database (it is otherwise not the case).

5.6.4 Notifications management

CubicWeb provides a machinery to ease notifications handling. To use it for a notification:

- write a view inheriting from *NotificationView*. The usual view api is used to generated the email (plain text) content, and additional `subject()` and `recipients()` methods are used to build the email's subject and recipients. *NotificationView* provides default implementation for both methods.
- write a hook for event that should trigger this notification, select the view (without rendering it), and give it to `cubicweb.hooks.notification.notify_on_commit()` so that the notification will be sent if the transaction succeed.

API details

class `cubicweb.subjects.notification.NotificationView`(*req=None, rset=None, **kwargs*)
abstract view implementing the “email” API (eg to simplify sending notification)

`cubicweb.hooks.notification.notify_on_commit`(*cnx, view, viewargs=None*)
register a notification view (see *NotificationView*) to be sent at post-commit time, ie only if the transaction has succeeded.
viewargs is an optional dictionary containing extra argument to be given to `render_and_send()`

5.6.5 Tasks

[WRITE ME]

- repository tasks

5.7 Tests

5.7.1 Unit tests

The *CubicWeb* framework provides the `cubicweb.devtools.testlib.CubicWebTC` test base class .

Tests shall be put into the `mycube/test` directory. Additional test data shall go into `mycube/test/data`.

It is much advised to write tests concerning entities methods, actions, hooks and operations, security. The *CubicWebTC* base class has convenience methods to help test all of this.

In the realm of views, automatic tests check that views are valid XHTML. See *Automatic views testing* for details.

Most unit tests need a live database to work against. This is achieved by CubicWeb using automatically sqlite (bundled with Python, see <http://docs.python.org/library/sqlite3.html>) as a backend.

The database is stored in the `mycube/test/tmpdb`, `mycube/test/tmpdb-template` files. If it does not (yet) exist, it will be built automatically when the test suite starts.

Warning: Whenever the schema changes (new entities, attributes, relations) one must delete these two files. Changes concerned only with entity or relation type properties (constraints, cardinalities, permissions) and generally dealt with using the `sync_schema_props_perms()` function of the migration environment do not need a database regeneration step.

Unit test by example

We start with an example extracted from the keyword cube (available from <https://forge.extranet.logilab.fr/cubicweb/cubes/keyword>).

```
from cubicweb.devtools.testlib import CubicWebTC
from cubicweb import ValidationError

class ClassificationHooksTC(CubicWebTC):

    def setup_database(self):
        with self.admin_access.repo_cnx() as cnx:
            group_etype = cnx.find('CWEType', name='CWGroup').one()
            c1 = cnx.create_entity('Classification', name=u'classif1',
                                  classifies=group_etype)
            user_etype = cnx.find('CWEType', name='CWUser').one()
            c2 = cnx.create_entity('Classification', name=u'classif2',
                                  classifies=user_etype)
            self.kwleid = cnx.create_entity('Keyword', name=u'kwgroup', included_in=c1).
            eid
            cnx.commit()

    def test_cannot_create_cycles(self):
        with self.admin_access.repo_cnx() as cnx:
            kw1 = cnx.entity_from_eid(self.kwleid)
            # direct obvious cycle
            with self.assertRaises(ValidationError):
                kw1.cw_set(subkeyword_of=kw1)
            cnx.rollback()
            # testing indirect cycles
            kw3 = cnx.execute('INSERT Keyword SK: SK name "kwgroup2", SK included_in C, '
                              'SK subkeyword_of K WHERE C name "classif1", K eid %(k)s'
                              {'k': kw1}).get_entity(0,0)
            kw3.cw_set(reverse_subkeyword_of=kw1)
            self.assertRaises(ValidationError, cnx.commit)
```

The test class defines a `setup_database()` method which populates the database with initial data. Each test of the class runs with this pre-populated database.

The test case itself checks that an Operation does its job of preventing cycles amongst Keyword entities.

The `create_entity` method of connection (or request) objects allows to create an entity. You can link this entity to other entities, by specifying as argument, the relation name, and the entity to link, as value. In the above example, the *Classification* entity is linked to a *CWEType* via the relation *classifies*. Conversely, if you are creating a *CWEType* entity, you can link it to a *Classification* entity, by adding *reverse_classifies* as argument.

Note: the `commit()` method is not called automatically. You have to call it explicitly if needed (notably to test operations). It is a good practice to regenerate entities with `entity_from_eid()` after a commit to avoid request cache effects.

You can see an example of security tests in the *Step 1: configuring security into the schema*.

It is possible to have these tests run continuously using `apycot`.

Managing connections or users

Since unit tests are done with the SQLITE backend and this does not support multiple connections at a time, you must be careful when simulating security, changing users.

By default, tests run with a user with admin privileges. Connections using these credentials are accessible through the `admin_access` object of the test classes.

The `repo_cnx()` method returns a connection object that can be used as a context manager:

```
# admin_access is a pre-cooked session wrapping object
# it is built with:
# self.admin_access = self.new_access('admin')
with self.admin_access.repo_cnx() as cnx:
    cnx.execute(...)
    self.create_user(cnx, login='user1')
    cnx.commit()

user1access = self.new_access('user1')
with user1access.web_request() as req:
    req.execute(...)
    req.cnx.commit()
```

On exit of the context manager, a rollback is issued, which releases the connection. Don't forget to issue the `cnx.commit()` calls!

Warning: Do not use references kept to the entities created with a connection from another one!

Email notifications tests

When running tests, potentially generated e-mails are not really sent but are found in the list `MAILBOX` of module `cubicweb.devtools.testlib`.

You can test your notifications by analyzing the contents of this list, which contains objects with two attributes:

- `recipients`, the list of recipients
- `msg`, `email.Message` object

Let us look at a simple example from the blog cube.

```
from cubicweb.devtools.testlib import CubicWebTC, MAILBOX

class BlogTestsCubicWebTC(CubicWebTC):
```

(continues on next page)

(continued from previous page)

```

"""test blog specific behaviours"""

def test_notifications(self):
    with self.admin_access.web_request() as req:
        cubicweb_blog = req.create_entity('Blog', title=u'cubicweb',
                                          description=u'cubicweb is beautiful')
        blog_entry_1 = req.create_entity('BlogEntry', title=u'hop',
                                         content=u'cubicweb hop')
        blog_entry_1.cw_set(entry_of=cubicweb_blog)
        blog_entry_2 = req.create_entity('BlogEntry', title=u'yes',
                                         content=u'cubicweb yes')
        blog_entry_2.cw_set(entry_of=cubicweb_blog)
        self.assertEqual(len(MAILBOX), 0)
        req.cnx.commit()
        self.assertEqual(len(MAILBOX), 2)
        mail = MAILBOX[0]
        self.assertEqual(mail.subject, '[data] hop')
        mail = MAILBOX[1]
        self.assertEqual(mail.subject, '[data] yes')

```

Visible actions tests

It is easy to write unit tests to test actions which are visible to a user or to a category of users. Let's take an example in the `conference cube`.

```

class ConferenceActionsTC(CubicWebTC):

    def setup_database(self):
        with self.admin_access.repo_cnx() as cnx:
            self.confeid = cnx.create_entity('Conference',
                                             title=u'my conf',
                                             url_id=u'conf',
                                             start_on=date(2010, 1, 27),
                                             end_on = date(2010, 1, 29),
                                             call_open=True,
                                             reverse_is_chair_at=chair,
                                             reverse_is_reviewer_at=reviewer).eid

    def test_admin(self):
        with self.admin_access.web_request() as req:
            rset = req.find('Conference').one()
            self.assertListEqual(self.pactions(req, rset),
                                [
                                    ('workflow', workflow.WorkflowActions),
                                    ('edit', confactions.ModifyAction),
                                    ('managepermission', actions.ManagePermissionsAction),
                                    ('addrelated', actions.AddRelatedActions),
                                    ('delete', actions.DeleteAction),
                                    ('generate_badge_action', badges.GenerateBadgeAction),
                                    ('addtalkinconf', confactions.
↪AddTalkInConferenceAction)
                                ])

```

(continues on next page)

(continued from previous page)

```

self.assertEqual(self.action_submenu(req, rset, 'addrelated'),
                 [(u'add Track in_conf Conference object',
                    u'http://testing.fr/cubicweb/add/Track'
                    u'?__linkto=in_conf%3A%(conf)s%3Asubject&'
                    u'__redirectpath=conference%2Fconf&'
                    u'__redirectvid=' % {'conf': self.confeid}),
                  ])

```

You just have to execute a rql query corresponding to the view you want to test, and to compare the result of `pactions()` with the list of actions that must be visible in the interface. This is a list of tuples. The first element is the action's `__regid__`, the second the action's class.

To test actions in a submenu, you just have to test the result of `action_submenu()` method. The last parameter of the method is the action's category. The result is a list of tuples. The first element is the action's title, and the second element the action's url.

5.7.2 Automatic views testing

This is done automatically with the `cubicweb.devtools.testlibAutomaticWebTest` class. At cube creation time, the `mycube/test/test_mycube.py` file contains such a test. The code here has to be uncommented to be usable, without further modification.

The `auto_populate` method uses a smart algorithm to create pseudo-random data in the database, thus enabling the views to be invoked and tested.

Depending on the schema, hooks and operations constraints, it is not always possible for the automatic `auto_populate` to proceed.

It is possible of course to completely redefine `auto_populate`. A lighter solution is to give hints (fill some class attributes) about what entities and relations have to be skipped by the `auto_populate` mechanism. These are:

- `no_auto_populate`, may contain a list of entity types to skip
- `ignored_relations`, may contain a list of relation types to skip
- `application_rql`, may contain a list of rql expressions that `auto_populate` cannot guess by itself; these must yield resultsets against which views may be selected.

Warning: Take care to not let the imported *AutomaticWebTest* in your test module namespace, else both your subclass *and* this parent class will be run.

5.7.3 Cache heavy database setup

Some test suites require a complex setup of the database that takes seconds (or even minutes) to complete. Doing the whole setup for each individual test makes the whole run very slow. The `CubicWebTC` class offer a simple way to prepare a specific database once for multiple tests. The `test_db_id` class attribute of your `CubicWebTC` subclass must be set to a unique identifier and the `pre_setup_database()` class method must build the cached content. As the `pre_setup_database()` method is not guaranteed to be called every time a test method is run, you must not set any class attribute to be used during test *there*. Databases for each `test_db_id` are automatically created if not already in cache. Clearing the cache is up to the user. Cache files are found in the `data/database` subdirectory of your test directory.

Warning: Take care to always have the same `pre_setup_database()` function for all classes with a given `test_db_id` otherwise your tests will have unpredictable results depending on the first encountered one.

5.7.4 Testing on a real-life database

The CubicWebTC class uses the `cubicweb.devtools.ApptestConfiguration` configuration class to setup its testing environment (database driver, user password, application home, and so on). The `cubicweb.devtools` module also provides a `RealDatabaseConfiguration` class that will read a regular cubicweb sources file to fetch all this information but will also prevent the database to be initialized and reset between tests.

For a test class to use a specific configuration, you have to set the `_config` class attribute on the class as in:

```
from cubicweb.devtools import RealDatabaseConfiguration
from cubicweb.devtools.testlib import CubicWebTC

class BlogRealDatabaseTC(CubicWebTC):
    _config = RealDatabaseConfiguration('blog',
                                       sourcefile='/path/to/realdb_sources')

    def test_blog_rss(self):
        with self.admin_access.web_request() as req:
            rset = req.execute('Any B ORDERBY D DESC WHERE B is BlogEntry, '
                              'B created_by U, U login "logilab", B creation_date D')
            self.view('rss', rset, req=req)
```

5.7.5 Testing with other cubes

Sometimes a small component cannot be tested all by itself, so one needs to specify other cubes to be used as part of the the unit test suite. This is handled by the `bootstrap_cubes` file located under `mycube/test/data`. One example from the *preview* cube:

```
card, file, preview
```

The format is:

- possibly several empty lines or lines starting with `#` (comment lines)
- one line containing a comma-separated list of cube names.

It is also possible to add a `schema.py` file in `mycube/test/data`, which will be used by the testing framework, therefore making new entity types and relations available to the tests.

5.7.6 Literate programming

CubicWeb provides some literate programming capabilities. The `cubicweb-ctl tool shell` command accepts different file formats. If your file ends with `.txt` or `.rst`, the file will be parsed by `doctest.testfile` with CubicWeb's *Migration* API enabled in it.

Create a `scenario.txt` file in the `test/` directory and fill with some content. Refer to the `doctest.testfile` [documentation](#).

Then, you can run it directly by:

```
$ cubicweb-ctl shell <cube_instance> test/scenario.txt
```

When your scenario file is ready, put it in a new test case to be able to run it automatically.

```
from os.path import dirname, join
from logilab.common.testlib import unittest_main
from cubicweb.devtools.testlib import CubicWebTC

class AcceptanceTC(CubicWebTC):

    def test_scenario(self):
        self.assertDocTestFile(join(dirname(__file__), 'scenario.txt'))

if __name__ == '__main__':
    unittest_main()
```

Skipping a scenario

If you want to set up initial conditions that you can't put in your unit test case, you have to use a `KeyboardInterrupt` exception only because of the way `doctest` module will catch all the exceptions internally.

```
>>> if condition_not_met:
...     raise KeyboardInterrupt('please, check your fixture.')
```

Passing parameters

Using extra arguments to parametrize your scenario is possible by prepending them by double dashes.

Please refer to the `cubicweb-ctl shell -help` usage.

Important: Your scenario file must be utf-8 encoded.

5.7.7 Test APIS

Using Pytest

The `pytest` utility (shipping with `logilab-common`, which is a mandatory dependency of CubicWeb) extends the Python unittest functionality and is the preferred way to run the CubicWeb test suites. Bare unittests also work the usual way.

To use it, you may:

- just launch `pytest` in your cube to execute all tests (it will discover them automatically)
- launch `pytest unittest_foo.py` to execute one test file
- launch `pytest unittest_foo.py bar` to execute all test methods and all test cases whose name contains `bar`

Additionally, the `-x` option tells pytest to exit at the first error or failure. The `-i` option tells pytest to drop into `pdb` whenever an exception occurs in a test.

When the `-x` option has been used and the run stopped on a test, it is possible, after having fixed the test, to relaunch pytest with the `-R` option to tell it to start testing again from where it previously failed.

Using the *TestCase* base class

The base class of CubicWebTC is `logilab.common.testlib.TestCase`, which provides a lot of convenient assertion methods.

class `logilab.common.testlib.TestCase`(*methodName: str* = 'runTest')

A unittest.TestCase extension with some additional methods.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

classmethod `datapath`(**fname: str*) → str

joins the object's datadir and *fname*

innerSkip(*msg: Optional[str]* = None) → `mypy_extensions.NoReturn`

mark a generative test as skipped for the <msg> reason

maxDiff = None

optval(*option, default=None*)

return the option value or default if the option is not define

set_description(*descr*)

sets the current test's description. This can be useful for generative tests because it allows to specify a description per yield

shortDescription() → `Optional[Any]`

override default unittest shortDescription to handle correctly generative tests

CubicWebTC API

class `cubicweb.devtools.testlib.CubicWebTC`(**args, **kwargs*)

abstract class for test using an apptest environment

attributes:

- *vreg*, the vregistry
- *schema*, self.vreg.schema
- *config*, cubicweb configuration
- *cnx*, repoapi connection to the repository using an admin user
- *session*, server side session associated to *cnx*
- *app*, the cubicweb publisher (for web testing)
- *repo*, the repository object
- *admlogin*, login of the admin user
- *admpassword*, password of the admin user
- *shell*, create and use shell environment
- *anonymous_allowed*: flag telling if anonymous browsing should be allowed

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

admin_request_from_url(*url*)

parses *url* and builds the corresponding CW-web request

req.form will be setup using the url's query string

property app

return a cubicweb publisher

assertSentEmail(*subject*, *recipients=None*, *nb_msgs=None*)

test recipients in system mailbox for given email subject

Parameters

- **subject** – email subject to find in mailbox
- **recipients** – list of email recipients
- **nb_msgs** – expected number of entries

Returns list of matched emails

configcls

alias of `cubicweb.devtools.ApptestConfiguration`

create_user(*req*, *login=None*, *groups=('users',)*, *password=None*, *email=None*, *commit=True*, ***kwargs*)

create and return a new user entity

ctrl_publish(*req*, *ctrl='edit'*, *rset=None*)

call the publish method of the edit controller

expect_redirect(*callback*, *req*)

call the given callback with *req* as argument, expecting to get a Redirect exception

expect_redirect_handle_request(*req*, *path='edit'*)

call the publish method of the application publisher, expecting to get a Redirect exception

static fake_form(*formid*, *field_dict=None*, *entity_field_dicts=()*)

Build `_cw.form` dictionary to fake posting of some standard cubicweb form

- *formid*, the form id, usually form's `__regid__`
- *field_dict*, dictionary of name:value for fields that are not tied to an entity
- *entity_field_dicts*, list of (entity, dictionary) where dictionary contains name:value for fields that are not tied to the given entity

http_publish(*url*, *data=None*)

like *url_publish*, except this returns a http response, even in case of errors. You may give form parameters using the *data* argument.

classmethod init_config(*config*)

configuration initialization hooks.

You may only want to override here the configuraton logic.

Otherwise, consider to use a different `ApptestConfiguration` defined in the *configcls* class attribute.

This method will be called by the database handler once the config has been properly bootstrapped.

list_actions_for(*rset*)

returns the list of actions that can be applied on *rset*

list_boxes_for(*rset*)

returns the list of boxes that can be applied on *rset*

list_startup_views()

returns the list of startup views

list_views_for(*rset*)

returns the list of views that can be applied on *rset*

new_access(*login*)

provide a new RepoAccess object for a given user

The access is automatically closed at the end of the test.

classmethod pre_setup_database(*cnx, config*)

add your pre database setup code by overriding this method

Do not forget to set the `cls.test_db_id` value to enable caching of the result.

remote_calling(*fname, *args, **kwargs*)

remote json call simulation

requestcls

alias of `cubicweb.devtools.fake.FakeRequest`

property schema

return the application schema

setUp()

Hook method for setting up the test fixture before exercising it.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

setup_database()

add your database setup code by overriding this method

tearDown()

Hook method for deconstructing the test fixture after testing it.

temporary_permissions(**perm_overrides, **perm_kwoverrides*)

Set custom schema permissions within context.

There are two ways to call this method, which may be used together :

- using positional argument(s):

```
rdef = self.schema['CWUser'].rdef('login')
with self.temporary_permissions((rdef, {'read': ()})):
    ...
```

- using named argument(s):

```
with self.temporary_permissions(CWUser={'read': ()}):
    ...
```

Usually the former will be preferred to override permissions on a relation definition, while the latter is well suited for entity types.

The allowed keys in the permission dictionary depend on the schema type (entity type / relation definition). Resulting permissions will be similar to `orig_permissions.update(partial_perms)`.

url_publish(*url*, *data*=None)

takes *url*, uses application's *app_resolver* to find the appropriate controller and result set, then publishes the result.

To simulate post of www-form-encoded data, give a *data* dictionary containing desired key/value associations.

This should pretty much correspond to what occurs in a real CW server except the apache-rewriter component is not called.

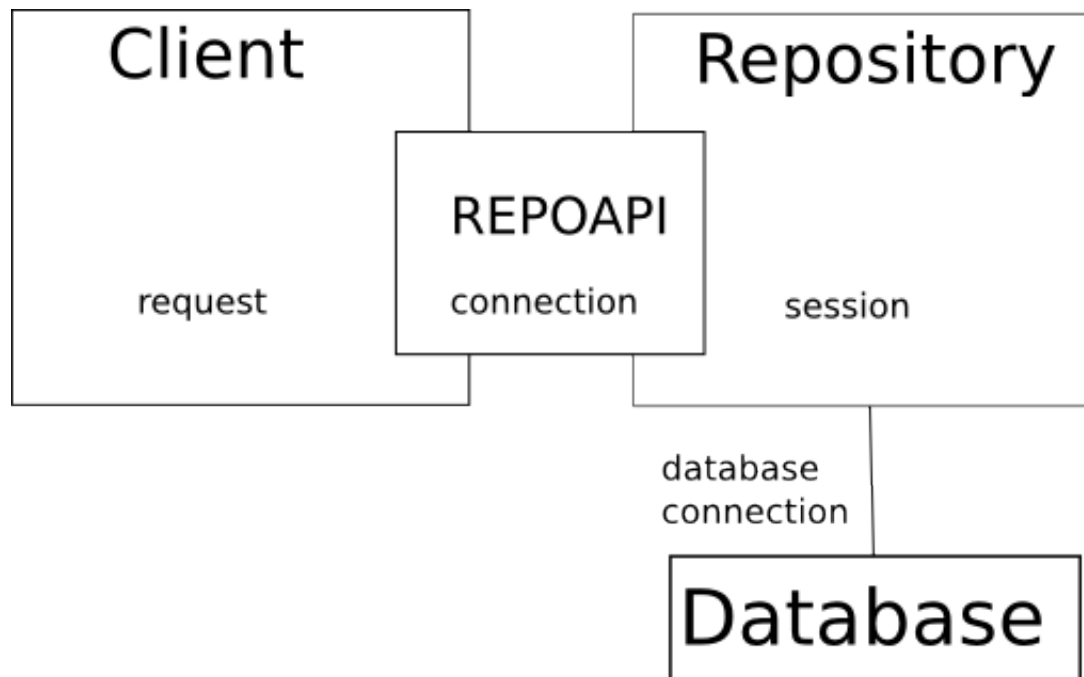
view(*vid*, *rset*=None, *req*=None, *template*='main-template', ***kwargs*)

This method tests the view *vid* on *rset* using *template*

If no error occurred while rendering the view, the HTML is analyzed and parsed.

Returns an instance of *cubicweb.devtools.htmlparser.PageInfo* encapsulation the generated HTML

5.7.8 What you need to know about request and session



First, remember to think that some code run on a client side, some other on the repository side. More precisely:

- client side: web interface, raw repoapi connection (cubicweb-ctl shell for instance);
- repository side: RQL query execution, that may trigger hooks and operation.

The client interacts with the repository through a repoapi connection.

Note: These distinctions are going to disappear in cubicweb 3.21 (if not before).

A repoapi connection is tied to a session in the repository. The connection and request objects are inaccessible from repository code / the session object is inaccessible from client code (theoretically at least).

The web interface provides a request class. That *request* object provides access to all cubicweb resources, eg:

- the registry (which itself provides access to the schema and the configuration);
- an underlying repoapi connection (when using `req.execute`, you actually call the repoapi);
- other specific resources depending on the client type (url generation according to base url, form parameters, etc.).

A *session* provides an api similar to a request regarding RQL execution and access to global resources (registry and all), but also has the following responsibilities:

- handle transaction data, that will live during the time of a single transaction. This includes the database connections that will be used to execute RQL queries.
- handle persistent data that may be used across different (web) requests
- security and hooks control (not possible through a request)

The `_cw` attribute

The `_cw` attribute available on every application object provides access to all cubicweb resources, i.e.:

- For code running on the client side (eg web interface view), `_cw` is a request instance.
- For code running on the repository side (hooks and operation), `_cw` is a Connection or Session instance.

Beware some views may be called with a session (e.g. notifications) or with a request.

Request, session and transaction

In the web interface, an HTTP request is handled by a single request, which will be thrown away once the response is sent.

The web publisher handles the transaction:

- commit / rollback is done automatically
- you should not commit / rollback explicitly, except if you really need it

Let's detail the process:

1. an incoming RQL query comes from a client to the web stack
2. the web stack opens an authenticated database connection for the request, which is associated to a user session
3. the query is executed (through the repository connection)
4. this query may trigger hooks. Hooks and operations may execute some rql queries through `cnx.execute`.
5. the repository gets the result of the query in 1. If it was a RQL read query, the database connection is released. If it was a write query, the connection is then tied to the session until the transaction is committed or rolled back.
6. results are sent back to the client

This implies several things:

- when using a request, or code executed in hooks, this database connection handling is totally transparent
- however, take care when writing tests: you are usually faking / testing both the server and the client side, so you have to decide when to use `RepoAccess.client_cnx` or `RepoAccess.repo_cnx`. Ask yourself "where will the code I want to test be running, client or repository side?". The response is usually: use a repo (since the "client connection" concept is going away in a couple of releases).

5.8 Migration

One of the main design goals of *CubicWeb* was to support iterative and agile development. For this purpose, multiple actions are provided to facilitate the improvement of an instance, and in particular to handle the changes to be applied to the data model, without loosing existing data.

The current version of a cube (and of cubicweb itself) is provided in the file `__pkginfo__.py` as a tuple of 3 integers.

5.8.1 Migration scripts management

Migration scripts has to be located in the directory *migration* of your cube and named accordingly:

```
<version n° X.Y.Z>[_<description>]_<mode>.py
```

in which :

- *X.Y.Z* is the model version number to which the script enables to migrate.
- *mode* (between the last “_” and the extension “.py”) is used for distributed installation. It indicates to which part of the application (RQL server, web server) the script applies. Its value could be :
 - *common*, applies to the RQL server as well as the web server and updates files on the hard drive (configuration files migration for example).
 - *web*, applies only to the web server and updates files on the hard drive.
 - *repository*, applies only to the RQL server and updates files on the hard drive.
 - *Any*, applies only to the RQL server and updates data in the database (schema and data migration for example).

Again in the directory *migration*, the file *depends.map* allows to indicate that for the migration to a particular model version, you always have to first migrate to a particular *CubicWeb* version. This file can contain comments (lines starting with #) and a dependency is listed as follows:

```
<model version n° X.Y.Z> : <cubicweb version n° X.Y.Z>
```

For example:

```
0.12.0: 2.26.0
0.13.0: 2.27.0
# 0.14 works with 2.27 <= cubicweb <= 2.28 at least
0.15.0: 2.28.0
```

5.8.2 Base context

The following identifiers are pre-defined in migration scripts:

- *config*, instance configuration
- *interactive_mode*, boolean indicating that the script is executed in an interactive mode or not
- *versions_map*, dictionary of migrated versions (key are cubes names, including ‘cubicweb’, values are (from version, to version))
- *confirm(question)*, function asking the user and returning true if the user answers yes, false otherwise (always returns true in non-interactive mode)

- `_()` is equivalent to `unicode` allowing to flag the strings to internationalize in the migration scripts.

In the *repository* scripts, the following identifiers are also defined:

- `commit(ask_confirm=True)`, request confirming and executing a “commit”
- `schema`, instance schema (readen from the database)
- `fsschema`, installed schema on the file system (e.g. schema of the updated model and cubicweb)
- `repo`, repository object
- `session`, repository session object

5.8.3 New cube dependencies

If your code depends on some new cubes, you have to add them in a migration script by using:

- `add_cube(cube, update_database=True)`, add a cube.
- `add_cubes(cubes, update_database=True)`, add a list of cubes.

The `update_database` parameter is telling if the database schema should be updated or if only the relevant persistent property should be inserted (for the case where a new cube has been extracted from an existing one, so the new cube schema is actually already in there).

If some of the added cubes are already used by an instance, they’ll simply be silently skipped.

To remove a cube use `drop_cube(cube, removedeps=False)`.

5.8.4 Schema migration

The following functions for schema migration are available in *repository* scripts:

- `add_attribute(etype, attrname, attrtype=None, commit=True)`, adds a new attribute to an existing entity type. If the attribute type is not specified, then it is extracted from the updated schema.
- `drop_attribute(etype, attrname, commit=True)`, removes an attribute from an existing entity type.
- `rename_attribute(etype, oldname, newname, commit=True)`, renames an attribute
- `add_entity_type(etype, auto=True, commit=True)`, adds a new entity type. If `auto` is `True`, all the relations using this entity type and having a known entity type on the other hand will automatically be added.
- `drop_entity_type(etype, commit=True)`, removes an entity type and all the relations using it.
- `rename_entity_type(oldname, newname, commit=True)`, renames an entity type
- `add_relation_type(rtype, addrdef=True, commit=True)`, adds a new relation type. If `addrdef` is `True`, all the relations definitions of this type will be added.
- `drop_relation_type(rtype, commit=True)`, removes a relation type and all the definitions of this type.
- `rename_relation_type(oldname, newname, commit=True)`, renames a relation type.
- `add_relation_definition(subjtype, rtype, objtype, commit=True)`, adds a new relation definition.
- `drop_relation_definition(subjtype, rtype, objtype, commit=True)`, removes a relation definition.
- `sync_schema_props_perms(etype=None, syncperms=True, syncprops=True, syncrdefs=True, commit=True)`, synchronizes properties and/or permissions on: - the whole schema if `etype` is `None` - an entity or relation type schema if `etype` is a string - a relation definition if `etype` is a 3-uple (subject, relation, object)

- *change_relation_props(subjtype, rtype, objtype, commit=True, **kwargs)*, changes properties of a relation definition by using the named parameters of the properties to change.
- *set_widget(etype, rtype, widget, commit=True)*, changes the widget used for the relation <rtype> of entity type <etype>.
- *set_size_constraint(etype, rtype, size, commit=True)*, changes the size constraints for the relation <rtype> of entity type <etype>.
- *update_bfss_path(old_path, new_path, commit=True)*, change the path from *old_path* to *new_path* in Bytes File-System Storage (bfss).

5.8.5 Data migration

The following functions for data migration are available in *repository* scripts:

- *rql(rql, kwargs=None, cachekey=None, ask_confirm=True)*, executes an arbitrary RQL query, either to interrogate or update. A result set object is returned.
- *add_entity(etype, *args, **kwargs)*, adds a new entity of the given type. The attribute and relation values are specified as named positional arguments.

5.8.6 Workflow creation

The following functions for workflow creation are available in *repository* scripts:

- *add_workflow(label, workflowof, initial=False, commit=False, **kwargs)*, adds a new workflow for a given type(s),
- *get_workflow_for(etype)*, return the workflow for the given entity type,
- *transition_by_name(self, trname)*, method of `cubicweb.entities.wfobjs.Workflow` instance that returns the transition named *trname*,
- *set_permissions(self, requiredgroups=(), conditions=(), reset=True)* method of `cubicweb.entities.wfobjs.Transition` instance that sets or adds (if *reset* is False) groups and conditions for this transition.

You can find more details about workflows in the chapter *Defining a Workflow* .

5.8.7 Configuration migration

The following functions for configuration migration are available in all scripts:

- *option_renamed(oldname, newname)*, indicates that an option has been renamed
- *option_group_change(option, oldgroup, newgroup)*, indicates that an option does not belong anymore to the same group.
- *option_added(option)*, indicates that an option has been added.
- *option_removed(option)*, indicates that an option has been deleted.

The *config* variable is an object which can be used to access the configuration values, for reading and updating, with a dictionary-like syntax.

Example 1: migration script changing the variable ‘sender-addr’ in *all-in-one.conf*. The script also checks that in that the instance is configured with a known value for that variable, and only updates the value in that case.

```
wrong_addr = 'cubicweb@loiglab.fr' # known wrong address
fixed_addr = 'cubicweb@logilab.fr'
configured_addr = config.get('sender-addr')
# check that the address has not been hand fixed by a sysadmin
if configured_addr == wrong_addr:
    config['sender-addr'] = fixed_addr
    config.save()
```

Example 2: checking the value of the database backend driver, which can be useful in case you need to issue backend-dependent raw SQL queries in a migration script.

```
dbdriver = config.sources()['system']['db-driver']
if dbdriver == "sqlserver2005":
    # this is now correctly handled by CW :-)
    sql('ALTER TABLE cw_Xxxx ALTER COLUMN cw_name varchar(64) NOT NULL;')
    commit()
else: # postgresql
    sync_schema_props_perms(ertype=('Xxxx', 'name', 'String'),
                             syncperms=False)
```

5.8.8 Others migration functions

Those functions are only used for low level operations that could not be accomplished otherwise or to repair damaged databases during interactive session. They are available in *repository* scripts:

- `sql(sql, args=None, ask_confirm=True)`, executes an arbitrary SQL query on the system source
- `add_entity_type_table(etype, commit=True)`
- `add_relation_type_table(rtype, commit=True)`
- `uninline_relation(rtype, commit=True)`

5.9 Profiling and performance

If you feel that one of your pages takes more time than it should to be generated, chances are that you're making too many RQL queries. Obviously, there are other reasons but experience tends to show this is the first thing to track down. Luckily, CubicWeb provides a configuration option to log RQL queries. In your `all-in-one.conf` file, set the **query-log-file** option:

```
# web application query log file
query-log-file=/home/user/myapp-rql.log
```

Then restart your application, reload your page and stop your application. The file `myapp-rql.log` now contains the list of RQL queries that were executed during your test. It's a simple text file containing lines such as:

```
Any A WHERE X eid %(x)s, X lastname A {'x': 448} -- (0.002 sec, 0.010 CPU sec)
Any A WHERE X eid %(x)s, X firstname A {'x': 447} -- (0.002 sec, 0.000 CPU sec)
```

The structure of each line is:

```
<RQL QUERY> <QUERY ARGS IF ANY> -- <TIME SPENT>
```

CubicWeb also provides the **exlog** command to examine and summarize data found in such a file:

```
$ cubicweb-ctl exlog /home/user/myapp-rql.log
0.07 50 Any A WHERE X eid %(x)s, X firstname A {}
0.05 50 Any A WHERE X eid %(x)s, X lastname A {}
0.01 1 Any X,AA ORDERBY AA DESC WHERE E eid %(x)s, E employees X, X modification_date AA
↪ {}
0.01 1 Any X WHERE X eid %(x)s, X owned_by U, U eid %(u)s {, }
0.01 1 Any B,T,P ORDERBY lower(T) WHERE B is Bookmark,B title T, B path P, B bookmarked_
↪ by U, U eid %(x)s {}
0.01 1 Any A,B,C,D WHERE A eid %(x)s,A name B,A creation_date C,A modification_date D {}
```

This command sorts and uniquifies queries so that it's easy to see where is the hot spot that needs optimization.

Do not neglect to set the **fetch_attrs** attribute you can define in your entity classes because it can greatly reduce the number of queries executed (see *Loaded attributes and default sorting management*).

You should also know about the **profile** option in the `all-in-on.conf`. If set, this option will make your application run in an **hotshot** session and store the results in the specified file.

Last but not least, if you're using the PostgreSQL database backend, VACUUMing your database can significantly improve the performance of the queries (by updating the statistics used by the query optimizer). Nowadays, this is done automatically from time to time, but if you've just imported a large amount of data in your db, you will want to vacuum it (with the `analyse` option on). Read the documentation of your database for more information.

5.10 Full Text Indexing in CubicWeb

When an attribute is tagged as *fulltext-indexable* in the datamodel, CubicWeb will automatically trigger hooks to update the internal fulltext index (i.e the `appears` SQL table) each time this attribute is modified.

CubicWeb also provides a `db-rebuild-fti` command to rebuild the whole fulltext on demand:

```
cubicweb@esope~$ cubicweb db-rebuild-fti my_tracker_instance
```

You can also rebuild the fulltext index for a given set of entity types:

```
cubicweb@esope~$ cubicweb db-rebuild-fti my_tracker_instance Ticket Version
```

In the above example, only fulltext index of entity types `Ticket` and `Version` will be rebuilt.

5.10.1 Standard FTI process

Considering an entity type ET, the default *fti* process is to :

1. fetch all entities of type ET
2. for each entity, adapt it to *IFTIndexable* (see *IFTIndexableAdapter*)
3. call `get_words()` on the adapter which is supposed to return a dictionary *weight -> list of words* as expected by `index_object()`. The tokenization of each attribute value is done by `tokenize()`.

See *IFTIndexableAdapter* for more documentation.

5.10.2 Yams and fulltext_container

It is possible in the datamodel to indicate that fulltext-indexed attributes defined for an entity type will be used to index not the entity itself but a related entity. This is especially useful for composite entities. Let's take a look at (a simplified version of) the base schema defined in CubicWeb (see `cubicweb.schemas.base`):

```
class CWUser(WorkflowableEntityType):
    login      = String(required=True, unique=True, maxsize=64)
    upassword  = Password(required=True)

class EmailAddress(EntityType):
    address = String(required=True, fulltextindexed=True,
                    indexed=True, unique=True, maxsize=128)

class use_email_relation(RelationDefinition):
    name = 'use_email'
    subject = 'CWUser'
    object = 'EmailAddress'
    cardinality = '*?'
    composite = 'subject'
```

The schema above states that there is a relation between `CWUser` and `EmailAddress` and that the `address` field of `EmailAddress` is fulltext indexed. Therefore, in your application, if you use fulltext search to look for an email address, CubicWeb will return the `EmailAddress` itself. But the objects we'd like to index are more likely to be the associated `CWUser` than the `EmailAddress` itself.

The simplest way to achieve that is to tag the `use_email` relation in the datamodel:

```
class use_email(RelationType):
    fulltext_container = 'subject'
```

5.10.3 Customizing how entities are fetched during db-rebuild-fti

`db-rebuild-fti` will call the `cw_fti_index_rql_limit()` class method on your entity type.

classmethod `AnyEntity.cw_fti_index_rql_limit(req, limit=1000)`

generate rsets of entities to FT-index

By default, each successive result set is limited to 1000 entities

5.10.4 Customizing get_words()

You can also customize the FTI process by providing your own `get_words()` implementation:

```
from cubicweb.entities.adapters import IFTIndexableAdapter

class SearchIndexAdapter(IFTIndexableAdapter):
    __regid__ = 'IFTIndexable'
    __select__ = is_instance('MyEntityClass')

    def fti_containers(self, _done=None):
        """this should yield any entity that must be considered to
```

(continues on next page)

(continued from previous page)

```

    fulltext-index self.entity

    CubicWeb's default implementation will look for yams'
    ``fulltex_container`` property.
    """

    yield self.entity
    yield self.entity.some_related_entity

def get_words(self):
    # implement any logic here
    # see http://www.postgresql.org/docs/9.1/static/textsearch-controls.html
    # for the actual signification of 'C'
    return {'C': ['any', 'word', 'I', 'want']}
```

5.11 Data Import

CubicWeb is designed to easily manipulate large amounts of data, and provides utilities to make imports simple.

The main entry point is `cubicweb.dataimport.importer` which defines an `ExtEntitiesImporter` class responsible for importing data from an external source in the form `ExtEntity` objects. An `ExtEntity` is a transitional representation of an entity to be imported in the *CubicWeb* instance; building this representation is usually domain-specific – e.g. dependent of the kind of data source (RDF, CSV, etc.) – and is thus the responsibility of the end-user.

Along with the importer, a *store* must be selected, which is responsible for insertion of data into the database. There exists different kind of *stores*, allowing to insert data within different levels of the *CubicWeb* API and with different speed/security tradeoffs. Those keeping all the *CubicWeb* hooks and security will be slower but the possible errors in insertion (bad data types, integrity error, ...) will be handled.

5.11.1 Example

Consider the following schema snippet.

```

class Person(EntityType):
    name = String(required=True)

class knows(RelationDefinition):
    subject = 'Person'
    object = 'Person'
```

along with some data in a `people.csv` file:

```

# uri,name,knowns
http://www.example.org/alice,Alice,
http://www.example.org/bob,Bob,http://www.example.org/alice
```

The following code (using a shell context) defines a function `extentities_from_csv` to read *Person* external entities coming from a CSV file and calls the `ExtEntitiesImporter` to insert corresponding entities and relations into the *CubicWeb* instance.

```

from cubicweb.dataimport import ucsvreader, RQLObjectStore
from cubicweb.dataimport.importer import ExtEntity, ExtEntitiesImporter

def extentities_from_csv(fpath):
    """Yield Person ExtEntities read from `fpath` CSV file."""
    with open(fpath) as f:
        for uri, name, knows in ucsvreader(f, skipfirst=True, skip_empty=False):
            yield ExtEntity('Person', uri,
                           {'name': set([name]), 'knows': set([knows])})

extenties = extentities_from_csv('people.csv')
store = RQLObjectStore(cnx)
importer = ExtEntitiesImporter(schema, store)
importer.import_entities(extenties)
commit()
rset = cnx.execute('String N WHERE X name N, X knows Y, Y name "Alice"')
assert rset[0][0] == u'Bob', rset

```

5.11.2 Importer API

Data import of external entities.

Main entry points:

```

class cubicweb.dataimport.importer.ExtEntitiesImporter(schema, store, extid2eid=None,
                                                       existing_relations=None,
                                                       etypes_order_hint=(), import_log=None,
                                                       raise_on_error=False)

```

This class is responsible for importing external entities, that is instances of [ExtEntity](#), into CubicWeb entities.

Parameters

- **schema** – the CubicWeb’s instance schema
- **store** – a CubicWeb *Store*
- **extid2eid** – optional {extid: eid} dictionary giving information on existing entities. It will be completed during import. You may want to use [cwuri2eid\(\)](#) to build it.
- **existing_relations** – optional {rtype: set((subj eid, obj eid))} mapping giving information on existing relations of a given type. You may want to use [RelationMapping](#) to build it.
- **etypes_order_hint** – optional ordered iterable on entity types, giving an hint on the order in which they should be attempted to be imported
- **import_log** – optional object implementing the SimpleImportLog interface to record events occurring during the import
- **raise_on_error** – optional boolean flag - default to false, indicating whether errors should be raised or logged. You usually want them to be raised during test but to be logged in production.

Instances of this class are meant to import external entities through [import_entities\(\)](#) which handles a stream of [ExtEntity](#). One may then plug arbitrary filters into the external entities stream.

import_entities(*ext_entities*)

Import given external entities (*ExtEntity*) stream (usually a generator).

class cubicweb.dataimport.importer.**ExtEntity**(*etype, extid, values=None*)

Transitional representation of an entity for use in data importer.

An external entity has the following properties:

- **extid** (external id), an identifier for the ext entity,
- **etype** (entity type), a string which must be the name of one entity type in the schema (eg. 'Person', 'Animal', ...),
- **values**, a dictionary whose keys are attribute or relation names from the schema (eg. 'first_name', 'friend'), and whose values are *sets*. For attributes of type Bytes, byte strings should be inserted in *values*.

For instance:

```
ext_entity.extid = 'http://example.org/person/debby'
ext_entity.etype = 'Person'
ext_entity.values = {'first_name': set([u"Deborah", u"Debby"]),
                    'friend': set(['http://example.org/person/john'])}
```

Utilities:

cubicweb.dataimport.importer.**cwuri2eid**(*cnx, etypes, source_eid=None*)

Return a dictionary mapping cwuri to eid for entities of the given entity types and / or source.

class cubicweb.dataimport.importer.**RelationMapping**(*cnx, source=None*)

Read-only mapping from relation type to set of related (subject, object) eids.

If *source* is specified, only returns relations implying entities from this source.

cubicweb.dataimport.importer.**use_extid_as_cwuri**(*extid2eid*)

Return a generator of *ExtEntity* objects that will set *cwuri* using entity's extid if the entity does not exist yet and has no *cwuri* defined.

extid2eid is an extid to eid dictionary coming from an *ExtEntitiesImporter* instance.

Example usage:

```
importer = ExtEntitiesImporter(cnx, store, import_log)
set_cwuri = use_extid_as_cwuri(importer.extid2eid)
importer.import_entities(set_cwuri(extentities))
```

Stores

Stores are responsible to insert properly formatted entities and relations into the database. They have the following API:

```
>>> user_eid = store.prepare_insert_entity('CWUser', login=u'johndoe')
>>> group_eid = store.prepare_insert_entity('CWUser', name=u'unknown')
>>> store.prepare_insert_relation(user_eid, 'in_group', group_eid)
>>> store.flush()
>>> store.commit()
>>> store.finish()
```


Some store **requires a flush** to copy data in the database, so if you want to have store independant code you should explicitly call it. (There may be multiple flushes during the process, or only one at the end if there is no memory issue). This is different from the commit which validates the database transaction. At last, the *finish()* method should be called in case the store requires additional work once everything is done.

- `prepare_insert_entity(<entity type>, **kwargs) -> eid`: given an entity type, attributes and in-lined relations, return the eid of the entity to be inserted, *with no guarantee that anything has been inserted in database*,
- `prepare_update_entity(<entity type>, eid, **kwargs) -> None`: given an entity type and eid, promise for update given attributes and inlined relations *with no guarantee that anything has been inserted in database*,
- `prepare_insert_relation(eid_from, rtype, eid_to) -> None`: indicate that a relation `rtype` should be added between entities with eids `eid_from` and `eid_to`. Similar to `prepare_insert_entity()`, *there is no guarantee that the relation will be inserted in database*,
- `flush()` -> `None`: flush any temporary data to database. May be called several times during an import,
- `commit()` -> `None`: commit the database transaction,
- `finish()` -> `None`: additional stuff to do after import is terminated.

`class cubicweb.dataimport.stores.NullStore`

Store that mainly describe the store API.

It may be handy to test input data files or to measure time taken by steps above the store (e.g. data parsing, importer, etc.): simply give a [NullStore](#) instance instead of the actual store.

Stores can also be used as context manager. If no exception is raised during the import, a final flush and the finish method are called. On the contrary, if something went wrong, we roll everything back.

`class cubicweb.dataimport.stores.RQLObjectStore(cnx)`

Store that works by making RQL queries, hence with all the cubicweb's machinery activated.

`class cubicweb.dataimport.stores.NoHookRQLObjectStore(cnx, metagen=None)`

Store that works by accessing low-level CubicWeb's source API, with all hooks deactivated. It may be given a metadata generator object to handle metadata which are usually handled by hooks.

Arguments: - `cnx`, a connection to the repository - `metagen`, optional [MetadataGenerator](#) instance

`class cubicweb.dataimport.stores.MetadataGenerator(cnx, baseurl=None, source=None, meta_skipped=())`

Class responsible for generating standard metadata for imported entities. You may want to derive it to add application specific's metadata. This class (or a subclass) may either be given to a nohook or massive store.

Parameters: * `cnx`: connection to the repository * `baseurl`: optional base URL to be used for `cwuri` generation - default to `config['base-url']` * `source`: optional source to be used as `cw_source` for imported entities

5.11.3 MassiveObjectStore

This store relies on *COPY FROM* sql commands to directly push data using SQL commands rather than using the whole *CubicWeb* API. For now, **it only works with PostgreSQL** as it requires the *COPY FROM* command. Anything related to CubicWeb (Hooks, for instance), are bypassed. It inserts entities directly by using one PostgreSQL *COPY FROM* query for a set of similarly structured entities.

This store is the fastest, if the table is small compared to the volume of data to insert. Indeed, it removes all indexes and constraints on the table before importing, and reapply them at the end. This means that if the table is small compared to the amount of data you want to insert, this store is better than the others.

NOTE: Because inlined¹ relations are stored in the entity's table, they must be set as any other attributes of the entity. For instance:

```
store.prepare_insert_entity("MyEType", name="toto", favorite_email=email_address.eid)
```

```
class cubicweb.dataimport.massive_store.MassiveObjectStore(cnx, slave_mode=False,
                                                           eids_seq_range=10000,
                                                           metagen=None, drop=True)
```

Store for massive import of data, with delayed insertion of meta data.

WARNINGS:

- This store may only be used with PostgreSQL for now, as it relies on the COPY FROM method, and on specific PostgreSQL tables to get all the indexes.
- This store can only insert relations that are not inlined (i.e., which do *not* have inlined=True in their definition in the schema), unless they are specified as entity attributes.

It should be used as follows:

```
store = MassiveObjectStore(cnx) eid_p = store.prepare_insert_entity('Person',
                                                                    cwuri=u'http://dbpedia.org/toto', name=u'Toto')

eid_loc = store.prepare_insert_entity('Location', cwuri=u'http://geonames.org/11111',
                                      name=u'Somewhere')

store.prepare_insert_relation(eid_p, 'lives_in', eid_loc) store.flush() ... store.commit()
store.finish()
```

Full-text indexation is not handled, you'll have to reindex the proper entity types by yourself if desired.

Create a MassiveObject store, with the following arguments:

- *cnx*, a connection to the repository
- *metagen*, optional MetadataGenerator instance
- *eids_seq_range*: size of eid range reserved by the store for each batch

5.12 Debug Channels

In *CubicWeb* 3.27 a new debug channels mechanism has been added to help build the pyramid debug toolbar custom panels. It isn't meant to do regular CW development but can be used for tools building (like the custom panel) if desired.

The API is really simple to use and is used like this:

```
from cubicweb.debug import subscribe_to_debug_channel, unsubscribe_to_debug_channel

# the callback will only receive one argument which is a python dict
# containing debug information
def example_debug_callback(message):
    print(message)
```

(continues on next page)

¹ An inlined relation is a relation defined in the schema with the keyword argument inlined=True. Such a relation is inserted in the database as an attribute of the entity whose subject it is.

(continued from previous page)

```
# "channel" must be one of: controller, rql, sql, vreg, registry_decisions
subscribe_to_debug_channel(channel, example_debug_callback)

# when it is not needed anymore (and to avoid dandling references)
unsubscribe_to_debug_channel(channel, example_debug_callback)
```

5.12.1 Channels documentation

The list of sent messages by channels:

Controller

This debug message will only be sent in a pyramid context. Emitted for each request.

```
{
  "kind": ctrlid,
  "request": request_object,
  "path": request_object.path,
  "controller": controller,
  "config": repo_configuration,
}
```

RQL

Emitted for each query.

```
{
  "rql": rql_as_a_string,
  # arguments used to format the query
  "args": args,
  # used to link rql and sql queries
  "rql_query_tracing_token": rql_query_tracing_token,
  "callstack": python_call_stack,
  "time": time_taken_in_ms_by_the_query,
  "result": the_result_as_python_data,
  "description": description_object,
}
```

SQL

Emitted for each query. Be advised that a SQL query generated by a RQL query will be emitted before the corresponding RQL query.

```
{
  "sql": sql_as_a_string,
  # arguments used to format the query
  "args": args,
```

(continues on next page)

(continued from previous page)

```
"rollback": True|False,
"callstack": "".join(traceback.format_stack()[::-1]),
# used to link rql and sql queries
"rql_query_tracing_token": rql_query_tracing_token,
"time": time_taken_in_ms_by_the_query,
}
```

vreg

This debug message will only be sent in a pyramid context. Emitted for each request.

```
{
  "vreg": vreg,
}
```

registry_decisions

This is emitted each time a decision is taken in a registry.

```
{
  "all_objects": [],
  "end_score": int,
  "winners": [],
  "winner": obj or None,
  "registry": obj,
  "args": args,
  "kwargs": kwargs,
}
```

5.13 API Reference

`cubicweb.debug.subscribe_to_debug_channel(channel, subscriber)`

`cubicweb.debug.unsubscribe_to_debug_channel(channel, subscriber)`

Unsubscribe a callable from a channel. It will raise `Exception` if the channel doesn't exist nor

5.14 Source connections pooler

CubicWeb comes with a connections pool for it's datasource (typically `sqlite` or `postgresql`), it is a dynamic pool meaning that:

- it will keep a minimum number of connections open (by default 0)
- when load increase it will open new connections
- if a max number of connections is set it will stop once it's reached
- if the max number of connections is zero, it is considered to be unlimited

- after some idle time (*connections-pool-idle-timeout*), if no new connections needed to be open on a new request, the pool will close one unused connection if the queue isn't empty
- if no connection are available after some time and the max number of connections has been reached, the connections pool will raise. To fix this, you can either increase the value of *connections-pool-max-size* or set it to 0 for an unlimited number of connections. A minimum of 5 connections per process is recommended if you want to set a max number.

Note that the connections pool won't be activated in some "quick start" situations like database dump/restore.

5.14.1 Configuration

The values used by the connections pool are fully configurable *in your instance configuration file* (usually the *all-in-one.conf*), here is the list:

- **connections-pooler-enabled**: enable the connections pooler, default: true. You want to disable the pool if you are using another external pooling system like pgbouncer.
- **connections-pool-max-size**: max size of the connections pool. 0 means unlimited. Each source supporting multiple connections will have this maximum number of opened connections, default: 0
- **connections-pool-min-size**: min size of the connections pool. Each source supporting multiple connections will have this minimum number of opened connections, default: 0
- **connections-pool-idle-timeout**: the delay, in seconds, after the last opened connection before which the pool will start closing unused connections. A connection is only closed on a request that didn't need to create a new connection, default: 600

WEB FRONTEND DEVELOPMENT

In this chapter, we will describe the core APIs for web development in the *CubicWeb* framework.

6.1 Publisher

What happens when an HTTP request is issued ?

The story begins with the `CubicWebPublisher.main_publish` method. We do not get upper in the bootstrap process because it is dependant on the used HTTP library.

What `main_publish` does:

- get a controller id and a result set from the path (this is actually delegated to the *urlpublisher* component)
- the controller is then selected (if not, this is considered an authorization failure and signaled as such) and called
- then either a proper result is returned, in which case the request/connection object issues a `commit` and returns the result
- or error handling must happen:
 - `ValidationErrors` pop up there and may lead to a redirect to a previously arranged url or standard error handling applies
 - an HTTP 500 error (*Internal Server Error*) is issued

Now, let's turn to the controller. There are many of them in `cubicweb.web.views.basecontrollers`. We can just follow the default *view* controller that is selected on a *view* path. See the [Controllers](#) chapter for more information on controllers.

The *View* controller's entry point is the *publish* method. It does the following:

- compute the *main* view to be applied, using either the given result set or building one from a user provided rql string (*rql* and *vid* can be forced from the url GET parameters), that is:
 - compute the *vid* using the result set and the schema (see `cubicweb.web.views.vid_from_rset`)
 - handle all error cases that could happen in this phase
- do some cache management chores
- select a main template (typically *TheMainTemplate*, see chapter [Templates](#))
- call it with the result set and the computed view.

What happens next actually depends on the template and the view, but in general this is the rendering phase.

6.1.1 CubicWebPublisher API

class cubicweb.web.application.CubicWebPublisher(*repo, config, session_handler_fact=<class 'cubicweb.web.application.CookieSessionHandler'>*)

the publisher is a singleton hold by the web frontend, and is responsible to publish HTTP request.

The http server will call its main entry point `application.handle_request`.

main_handle_request(*req*)

Process an HTTP request *req*

Parameters *req* (*web.Request*) – the request object

It returns the content of the http response. HTTP header and status are set on the Request object.

You have to provide both a repository and web-server config at initialization. In all in one instance both config will be the same.

core_handle(*req*)

method called by the main publisher to process <req> relative path

should return a string containing the resulting page or raise a *NotFound* exception

Parameters *req* (*web.Request*) – the request object

Return type str

Returns the result of the published url

critical(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

`logger.critical("Houston, we have a %s", "major disaster", exc_info=1)`

debug(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

`logger.debug("Houston, we have a %s", "thorny problem", exc_info=1)`

error(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

`logger.error("Houston, we have a %s", "major problem", exc_info=1)`

exception(*msg, *args, exc_info=True, **kwargs*)

Convenience method for logging an ERROR with exception information.

get_session(*req*)

Return a session object corresponding to credentials held by the req

May raise *AuthenticationError*.

info(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

`logger.info("Houston, we have a %s", "interesting problem", exc_info=1)`

log_handle_request(*req*)

wrapper around `_publish` to log all queries executed for a given accessed path

main_handle_request(*req*)

Process an HTTP request *req*

Parameters **req** (*web.Request*) – the request object

It returns the content of the http response. HTTP header and status are set on the Request object.

redirect_handler(*req, ex*)

handle redirect - comply to *ex* status - set header field - return empty content

warning(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument `exc_info` with a true value, e.g.

`logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)`

6.2 Controllers

6.2.1 Overview

Controllers are responsible for taking action upon user requests (loosely following the terminology of the MVC meta pattern).

The following controllers are provided out-of-the box in CubicWeb. We list them by category. They are all defined in (`cubicweb.web.views.basecontrollers`).

Browsing:

- the View controller is associated with most browsing actions within a CubicWeb application: it always instantiates a *TheMainTemplate* and lets the ResultSet/Views dispatch system build up the whole content; it handles `ObjectNotFound` and `NoSelectableObject` errors that may bubble up to its entry point, in an end-user-friendly way (but other programming errors will slip through)
- the `JSonpController` is a wrapper around the `ViewController` that provides `jsonp` services. Padding can be specified with the `callback` request parameter. Only `jsonexport` / `ejsonexport` views can be used. If another `vid` is specified, it will be ignored and replaced by `jsonexport`. Request is anonymized to avoid returning sensitive data and reduce the risks of CSRF attacks;
- the Login/Logout controllers make effective user login or logout requests

Edition:

- the Edit controller (see *The edit controller*) handles CRUD operations in response to a form being submitted; it works in close association with the Forms, to which it delegates some of the work
- the `Form validator controller` provides form validation from Ajax context, using the Edit controller, to implement the classic form handling loop (user edits, hits *submit/apply*, validation occurs server-side by way of the Form validator controller, and the UI is decorated with failure information, either global or per-field , until it is valid)

Other:

- the `SendMail controller` (`web/views/basecontrollers.py`) is responsible for outgoing email notifications
- the `MailBugReport controller` (`web/views/basecontrollers.py`) allows to quickly have a *reportbug* feature in one's application

- the `cubicweb.web.views.ajaxcontroller.AjaxController` (`cubicweb.web.views.ajaxcontroller`) provides services for Ajax calls, typically using JSON as a serialization format for input, and sometimes using either JSON or XML for output. See [Ajax](#) chapter for more information.

6.2.2 Registration

All controllers (should) live in the ‘controllers’ namespace within the global registry.

6.2.3 Concrete controllers

Most API details should be resolved by source code inspection, as the various controllers have differing goals. See for instance the [The edit controller](#) chapter.

`cubicweb.web.controller` contains the top-level abstract Controller class and its unimplemented entry point `publish(rset=None)` method.

A handful of helpers are also provided there:

- `process_rql` builds a result set from an rql query typically issued from the browser (and available through `_cw.form['rql']`)
- `validate_cache` will force cache validation handling with respect to the HTTP Cache directives (that were typically originally issued from a previous server -> client response); concrete Controller implementations dealing with HTTP (thus, for instance, not the SendMail controller) may very well call this in their publication process.

6.3 The Request class (`cubicweb.web.request`)

6.3.1 Overview

A request instance is created when an HTTP request is sent to the web server. It contains informations such as form parameters, authenticated user, etc. It is a very prevalent object and is used throughout all of the framework and applications, as you’ll access to almost every resources through it.

A request represents a user query, either through HTTP or not (we also talk about RQL queries on the server side for example).

Here is a non-exhaustive list of attributes and methods available on request objects (grouped by category):

- *Browser control:*
 - `ie_browser`: tells if the browser belong to the Internet Explorer family
- *User and identification:*
 - `user`, instance of `cubicweb.entities.authobjs.CWUser` corresponding to the authenticated user
- *Session data handling*
 - `session.data` is the dictionary of the session data; it can be manipulated like an ordinary Python dictionary
- *Edition (utilities for edition control):*
 - `cancel_edition`: resets error url and cleans up pending operations
 - `create_entity`: utility to create an entity (from an etype, attributes and relation values)
 - `datadir_url`: returns the url to the merged external resources (*CubicWeb*’s `web/data` directory plus all `data` directories of used cubes)

- *edited_eids*: returns the list of eids of entities that are edited under the current http request
- *eid_rset(eid)*: utility which returns a result set from an eid
- *entity_from_eid(eid)*: returns an entity instance from the given eid
- *encoding*: returns the encoding of the current HTTP request
- *ensure_ro_rql(rql)*: ensure some rql query is a data request
- *etype_rset*
- *form*, dictionary containing the values of a web form
- *encoding*, character encoding to use in the response
- *HTTP*
 - *authmode*: returns a string describing the authentication mode (http, cookie, ...)
 - *lang*: returns the user agents/browser's language as carried by the http request
 - *demote_to_html()*: in the context of an XHTML compliant browser, this will force emission of the response as an HTML document (using the http content negotiation)
- *Cookies handling*
- *get_cookie()*, returns a dictionary containing the value of the header HTTP 'Cookie'
- *set_cookie(cookie, key, maxage=300)*, adds a header HTTP *Set-Cookie*, with a minimal 5 minutes length of duration by default (*maxage = None* returns a *session* cookie which will expire when the user closes the browser window)
- *remove_cookie(cookie, key)*, forces a value to expire
- *URL handling*
 - *build_url(__vid, *args, **kwargs)*: return an absolute URL using params dictionary key/values as URL parameters. Values are automatically URL quoted, and the publishing method to use may be specified or will be guessed.
 - *build_url_params(**kwargs)*: returns a properly prepared (quoted, separators, ...) string from the given parameters
 - *url()*, returns the full URL of the HTTP request
 - *base_url()*, returns the root URL of the web application
 - *relative_path()*, returns the relative path of the request
- *Web resource (.css, .js files, etc.) handling*:
 - *add_css(cssfiles)*: adds the given list of css resources to the current html headers
 - *add_js(jsfiles)*: adds the given list of javascript resources to the current html headers
 - *add_onload(jscode)*: inject the given jscode fragment (a unicode string) into the current html headers, wrapped inside a *document.ready(...)* or another ajax-friendly one-time trigger event
 - *add_header(header, values)*: adds the header/value pair to the current html headers
 - *status_out*: control the HTTP status of the response
- *And more...*
 - *set_content_type(content_type, filename=None)*, adds the header HTTP 'Content-Type'
 - *get_header(header)*, returns the value associated to an arbitrary header of the HTTP request

- `set_header(header, value)`, adds an arbitrary header in the response
- `execute(*args, **kwargs)`, executes an RQL query and return the result set
- `property_value(key)`, properties management (*CWProperty*)
- dictionary *data* to store data to share informations between components *while a request is executed*

Please note that this class is abstract and that a concrete implementation will be provided by the *frontend* web used. For the views or others that are executed on the server side, most of the interface of *Request* is defined in the session associated to the client.

6.3.2 API

The elements we gave in overview for above are built in three layers, from `cubicweb.req.RequestSessionBase`, `cubicweb.repoapi.Connection` and `cubicweb.web.ConnectionCubicWebRequestBase`.

class `cubicweb.req.RequestSessionBase(vreg)`

base class containing stuff shared by server session and web request

request/session is the main resources accessor, mainly through it's *vreg* attribute:

Attribute *vreg* the instance's registry

Attribute *vreg.schema* the instance's schema

Attribute *vreg.config* the instance's configuration

base_url()

Return the root url of the instance.

build_url(*args, **kwargs)

return an absolute URL using params dictionary key/values as URL parameters. Values are automatically URL quoted, and the publishing method to use may be specified or will be guessed.

raises *ValueError* if *None* is found in arguments

build_url_params(kwargs)**

return encoded params to incorporate them in a URL

build_url_path(method, kwargs)

return the “path” part of an URL

create_entity(etype, **kwargs)

add a new entity of the given type

Example (in a shell session):

```
>>> c = create_entity('Company', name=u'Logilab')
>>> create_entity('Person', firstname=u'John', surname=u'Doe',
...               works_for=c)
```

eid_rset(eid, etype=None)

return a result set for the given *eid* without doing actual query (we have the *eid*, we can suppose it exists and user has access to the entity)

empty_rset()

return a guaranteed empty result

ensure_ro_rql(*rql*)

raise an exception if the given rql is not a select query

entity_from_eid(*eid*, *etype*=None)

return an entity instance for the given eid. No query is done

etype_rset(*etype*, *size*=1)

return a fake result set for a particular entity type

exists(*etype*, ***kwargs*)

return if it exists at least one entity of the given type and attribute values.

```
>>> is_stallman_there = cnx.exists('CWUser', login=u"rms")
>>> # more optimized than
>>> is_stallman_there = bool(cnx.find('CWUser', login=u"rms"))
>>> # it did a full scan table :-/
```

find(*etype*, *limit*=None, ***kwargs*)

find entities of the given type and attribute values.

Parameters

- **etype** – the type of entities to return.
- **limit** – the max number of entities to return.

Returns A ResultSet

```
>>> users = find('CWGroup', name=u"users").one()
>>> groups = find('CWGroup').entities()
```

format_date(*date*, *date_format*=None, *time*=False)

return a string for a date time according to instance's configuration

format_float(*num*)

return a string for floating point number according to instance's configuration

format_time(*time*)

return a string for a time according to instance's configuration

parse_datetime(*value*, *etype*='Datetime')

get a datetime or time from a string (according to etype) Datetime formatted as Date are accepted

printable_value(*attrtype*, *value*, *props*=None, *displaytime*=True, *formatters*={ 'BigInt': <function print_int>, 'Boolean': <function print_boolean>, 'Bytes': <function print_bytes>, 'Date': <function print_date>, 'Datetime': <function print_datetime>, 'Decimal': <function print_float>, 'Float': <function print_float>, 'Int': <function print_int>, 'Interval': <function print_timedelta>, 'String': <function print_string>, 'TZDatetime': <function print_tzdatetime>, 'TZTime': <function print_tztime>, 'Time': <function print_time>})

return a displayable value (i.e. unicode string)

property_value(*key*)

return value of the property with the given key, giving priority to user specific value if any, else using site value

rebuild_url(*url*, ***newparams*)

return the given url with newparams inserted. If any new params is already specified in the url, it's overridden by the new value

newparams may only be mono-valued.

set_language(*lang*)

install i18n configuration for *lang* translation.

Raises `KeyError` if translation doesn't exist.

url_parse_qsl(*querystring*)

return a list of (key, val) found in the url quoted query string

url_quote(*value*, *safe=""*)

urllib.quote is not unicode safe, use this method to do the necessary encoding / decoding. Also it's designed to quote each part of a url path and so the '/' character will be encoded as well.

url_unquote(*quoted*)

returns a unicode unquoted string

decoding is based on *self.encoding* which is the encoding used in *url_quote*

user_data(***kwargs*)

returns a dictionary with this user's information.

The keys are :

login The user login

name The user name, returned by `user.name()`

email The user principal email

view(*_RequestSessionBase__vid*, *rset=None*, *_RequestSessionBase__fallback_oid=None*, *_RequestSessionBase__registry='views'*, *initargs=None*, *w=None*, ***kwargs*)

Select object with the given id (*__oid*) then render it. If the object isn't selectable, try to select fallback object if *__fallback_oid* is specified.

If specified *initargs* is expected to be a dictionary containing arguments that should be given to selection (hence to object's *__init__* as well), but not to *render()*. Other arbitrary keyword arguments will be given to selection *and* to *render()*, and so should be handled by object's *call* or *cell_call* method..

class `cubicweb.repoapi.Connection`(*repo*, *user*)

Repository Connection

Holds all connection related data

Database connection resources:

hooks_in_progress, boolean flag telling if the executing query is coming from a repoapi connection or is a query from within the repository (e.g. started by hooks)

cnxset, the connections set to use to execute queries on sources. If the transaction is read only, the connection set may be freed between actual queries. This allows multiple connections with a reasonably low connection set pool size.

mode, string telling the connections set handling mode, may be one of 'read' (connections set may be freed), 'write' (some write was done in the connections set, it can't be freed before end of the transaction), 'transaction' (we want to keep the connections set during all the transaction, with or without writing)

Shared data:

`data` is a dictionary bound to the underlying session, who will be present for the life time of the session. This may be useful for web clients that rely on the server for managing bits of session-scoped data.

`transaction_data` is a dictionary cleared at the end of the transaction. Hooks and operations may put arbitrary data in there.

Internal state:

`pending_operations`, ordered list of operations to be processed on commit/rollback

`commit_state`, describing the transaction commit state, may be one of None (not yet committing), 'precommit' (calling precommit event on operations), 'postcommit' (calling postcommit event on operations), 'uncommitable' (some `ValidationError` or `Unauthorized` error has been raised during the transaction and so it must be rolled back).

Hooks controls:

deny_all_hooks_but(*categories)

Context manager to disable all hooks but those in the given categories.

allow_all_hooks_but(*categories)

Context manager to enable all hooks but those in the given categories.

Security level Management:

`read_security` and `write_security`, boolean flags telling if read/write security is currently activated.

add_operation(operation, index=None)

add an operation to be executed at the end of the transaction

add_relation(fromeid, rtype, toeid)

provide direct access to the repository method to add a relation.

This is equivalent to the following rql query:

SET X rtype Y WHERE X eid fromeid, T eid toeid

without read security check but also all the burden of rql execution. You may use this in hooks when you know both eids of the relation you want to add.

add_relations(relations)

set many relation using a shortcut similar to the one in `add_relation`

`relations` is a list of 2-uples, the first element of each 2-uple is the `rtype`, and the second is a list of (`fromeid`, `toeid`) tuples

added_in_transaction(eid)

return True if the entity of the given `eid` is being created in the current transaction

allow_all_hooks_but(*categories)

Context manager to enable all hooks but those in the given categories.

cached_entities()

return the whole entity cache

clear()

reset internal data

commit()

commit the current session's transaction

commit_state

(None, 'precommit', 'postcommit', 'uncommitable')

critical(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=1)

debug(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

logger.debug("Houston, we have a %s", "thorny problem", exc_info=1)

delete_relation(*fromeid*, *rtype*, *toeid*)

provide direct access to the repository method to delete a relation.

This is equivalent to the following rql query:

```
DELETE X rtype Y WHERE X eid fromeid, T eid toeid
```

without read security check but also all the burden of rql execution. You may use this in hooks when you know both eids of the relation you want to delete.

deleted_in_transaction(*eid*)

return True if the entity of the given *eid* is being deleted in the current transaction

deny_all_hooks_but(**categories*)

Context manager to disable all hooks but those in the given categories.

drop_entity_cache()

Drop the whole entity cache.

entity_cache(*eid*)

get cache entity for *eid*

entity_type(*eid*)

Return entity type for the entity with id *eid*.

error(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

logger.error("Houston, we have a %s", "major problem", exc_info=1)

exception(*msg*, **args*, *exc_info=True*, ***kwargs*)

Convenience method for logging an ERROR with exception information.

execute(*rql*, *kwargs=None*, *build_descr=True*)

db-api like method directly linked to the querier execute method.

See `cubicweb.dbapi.Cursor.execute()` documentation.

get_option_value(*option*)

Return the value for *option* in the configuration.

get_schema()

Return the schema currently used by the repository.

info(*msg*, **args*, ***kwargs*)

Log ‘msg % args’ with severity ‘INFO’.

To pass exception information, use the keyword argument `exc_info` with a true value, e.g.

`logger.info(“Houston, we have a %s”, “interesting problem”, exc_info=1)`

is_hook_activated(*hook*)

return a boolean telling if the given hook class is currently activated or not

is_hook_category_activated(*category*)

return a boolean telling if the given category is currently activated or not

pending_operations

ordered list of operations to be processed on commit/rollback

repo

server.Repository object

rollback()

rollback the current transaction

running_hooks_ops()

this context manager should be called whenever hooks or operations are about to be run (but after hook selection)

It will help the undo logic record pertinent metadata or some hooks to run (or not) depending on who/what issued the query.

set_entity_cache(*entity*)

Add *entity* to the connection entity cache

system_sql(*sql*, *args=None*, *rollback_on_failure=True*, *rql_query_tracing_token=None*)

return a sql cursor on the system database

transaction_actions(*txuuid*, *public=True*)

Return an ordered list of actions effectuated during that transaction.

If *public* is true, return only ‘public’ actions, i.e. not ones triggered under the cover by hooks, else return all actions.

raise *NoSuchTransaction* if the transaction is not found or if the user is not allowed (eg not in managers group).

transaction_data

dict containing arbitrary data cleared at the end of the transaction

transaction_info(*txuuid*)

Return transaction object for the given uid.

raise *NoSuchTransaction* if not found or if session’s user is not allowed (eg not in managers group and the transaction doesn’t belong to him).

undo_transaction(*txuuid*)

Undo the given transaction. Return potential restoration errors.

raise *NoSuchTransaction* if not found or if user is not allowed (eg not in managers group).

undoable_transactions(*ueid=None, **actionfilters*)

Return a list of undoable transaction objects by the connection's user, ordered by descendant transaction time.

Managers may filter according to user (*eid*) who has done the transaction using the *ueid* argument. Others will only see their own transactions.

Additional filtering capabilities is provided by using the following named arguments:

- *etype* to get only transactions creating/updating/deleting entities of the given type
- *eid* to get only transactions applied to entity of the given *eid*
- *action* to get only transactions doing the given action (action in 'C', 'U', 'D', 'A', 'R'). If *etype*, action can only be 'C', 'U' or 'D'.
- *public*: when additional filtering is provided, they are by default only searched in 'public' actions, unless a *public* argument is given and set to false.

warning(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)

class cubicweb.web.request.ConnectionCubicWebRequestBase(*vreg, form=None, headers={}*)

6.4 RQL search bar

The RQL search bar is a visual component, hidden by default, the tiny *search* input being enough for common use cases.

An autocompletion helper is provided to help you type valid queries, both in terms of syntax and in terms of schema validity.

class cubicweb.web.views.magicsearch.RQLSuggestionsBuilder(**args, **kwargs*)

This component was previously used by the *rql_suggest* ajax function to provide completion to the search bar. It is now deprecated in favor of the `cubicweb.rqlsuggestions.RQLSuggestionsBuilder` class.

build_suggestions(*user_rql*)

6.4.1 How search is performed

You can use the *rql search bar* to either type RQL queries, plain text queries or standard shortcuts such as `<EntityType>` or `<EntityType> <attrname> <value>`.

Ultimately, all queries are translated to rql since it's the only language understood on the server (data) side. To transform the user query into RQL, CubicWeb uses the so-called *magicsearch component*, defined in `cubicweb.web.views.magicsearch`, which in turn delegates to a number of query preprocessor that are responsible of interpreting the user query and generating corresponding RQL.

The code of the main processor loop is easy to understand:

```
for proc in self.processors:
    try:
        return proc.process_query(uquery, req)
```

(continues on next page)

(continued from previous page)

```
except (RQLSyntaxError, BadRQLQuery):
    pass
```

The idea is simple: for each query processor, try to translate the query. If it fails, try with the next processor, if it succeeds, we're done and the RQL query will be executed.

6.5 The View system

This chapter aims to describe the concept of a *view* used all along the development of a web application and how it has been implemented in *CubicWeb*.

6.5.1 Principles

We'll start with a description of the interface providing a basic understanding of the available classes and methods, then detail the view selection principle.

A *View* is an object responsible for the rendering of data from the model into an end-user consummable form. They typically churn out an XHTML stream, but there are views concerned with email other non-html outputs.

Discovering possible views

It is possible to configure the web user interface to have a left box showing all the views than can be applied to the current result set.

To enable this, click on your login at the top right corner. Chose "user preferences", then "boxes", then "possible views box" and check "visible = yes" before validating your changes.

The views listed there we either not selected because of a lower score, or they were deliberately excluded by the main template logic.

Basic class for views

Class View

```
class cubicweb.view.View(req=None, rset=None, **kwargs)
```

This class is an abstraction of a view class, used as a base class for every renderable object such as views, templates and other user interface components.

A *View* is instantiated to render a result set or part of a result set. *View* subclasses may be parametrized using the following class attributes:

templatable indicates if the view may be embedded in a main template or if it has to be rendered standalone (i.e. pure XML views must not be embedded in the main template of HTML pages)

content_type if the view is not templatable, it should set the *content_type* class attribute to the correct MIME type (text/xhtml being the default)

category this attribute may be used in the interface to regroup related objects (view kinds) together

paginable

binary

A view writes to its output stream thanks to its attribute *w* (the *append* method of an *UStreamIO*, except for binary views).

At instantiation time, the standard *_cw*, and *cw_rset* attributes are added and the *w* attribute will be set at rendering time to a write function to use.

The basic interface for views is as follows (remember that the result set has a tabular structure with rows and columns, hence cells):

- *render(**context)*, render the view by calling *call* or *cell_call* depending on the context
- *call(**kwargs)*, call the view for a complete result set or null (the default implementation calls *cell_call()* on each cell of the result set)
- *cell_call(row, col, **kwargs)*, call the view for a given cell of a result set (*row* and *col* being integers used to access the cell)
- *url()*, returns the URL enabling us to get the view with the current result set
- *wview(__vid, rset, __fallback_vid=None, **kwargs)*, call the view of identifier *__vid* on the given result set. It is possible to give a fallback view identifier that will be used if the requested view is not applicable to the result set.
- *html_headers()*, returns a list of HTML headers to be set by the main template
- *page_title()*, returns the title to use in the HTML header *title*

Other basic view classes

Here are some of the subclasses of *View* defined in *cubicweb.view* that are more concrete as they relate to data rendering within the application:

```
class cubicweb.view.EntityView(req=None, rset=None, **kwargs)
```

base class for views applying on an entity (i.e. uniform result set)

```
class cubicweb.view.StartupView(req=None, rset=None, **kwargs)
```

base class for views which doesn't need a particular result set to be displayed (so they can always be displayed!)

```
class cubicweb.view.EntityStartupView(req, rset=None, **kwargs)
```

base class for entity views which may also be applied to None result set (usually a default *rql* is provided by the view class)

```
class cubicweb.view.AnyRsetView(req=None, rset=None, **kwargs)
```

base class for views applying on any non empty result sets

Examples of views class

- Using *templatable*, *content_type* and HTTP cache configuration

```
class RSSView(XMLView):
    __regid__ = 'rss'
    title = _('rss')
    templatable = False
    content_type = 'text/xml'
    http_cache_manager = MaxAgeHTTPCacheManager
    cache_max_age = 60*60*2 # stay in http cache for 2 hours by default
```

- Using a custom selector

```
class SearchForAssociationView(EntityView):
    """view called by the edition view when the user asks
    to search for something to link to the edited eid
    """
    __regid__ = 'search-associate'
    title = _('search for association')
    __select__ = one_line_rset() & match_search_state('linksearch') & is_instance('Any')
```

XML views, binaries views...

For views generating other formats than HTML (an image generated dynamically for example), and which can not simply be included in the HTML page generated by the main template (see above), you have to:

- set the attribute *templatable* of the class to *False*
- set, through the attribute *content_type* of the class, the MIME type generated by the view to *application/octet-stream* or any relevant and more specialised mime type

For views dedicated to binary content creation (like dynamically generated images), we have to set the attribute *binary* of the class to *True* (which implies that *templatable* == *False*, so that the attribute *w* of the view could be replaced by a binary flow instead of unicode).

6.5.2 Templates

Templates are the entry point for the *CubicWeb* view system. As seen in *Discovering possible views*, there are two kinds of views: the templatable and non-templatable.

Non-templatable views

Non-templatable views are standalone. They are responsible for all the details such as setting a proper content type (or mime type), the proper document headers, namespaces, etc. Examples are pure xml views such as RSS or Semantic Web views (*SIOC*, *DOAP*, *FOAF*, *Linked Data*, etc.), and views which generate binary files (pdf, excel files, etc.)

To notice that a view is not templatable, you just have to set the view's class attribute *templatable* to *False*. In this case, it should set the *content_type* class attribute to the correct MIME type. By default, it is text/xhtmll. Additionally, if your view generate a binary file, you have to set the view's class attribute *binary* to *True* too.

Templatable views

Templatable views are not concerned with such pesky details. They leave it to the template. Conversely, the template's main job is to:

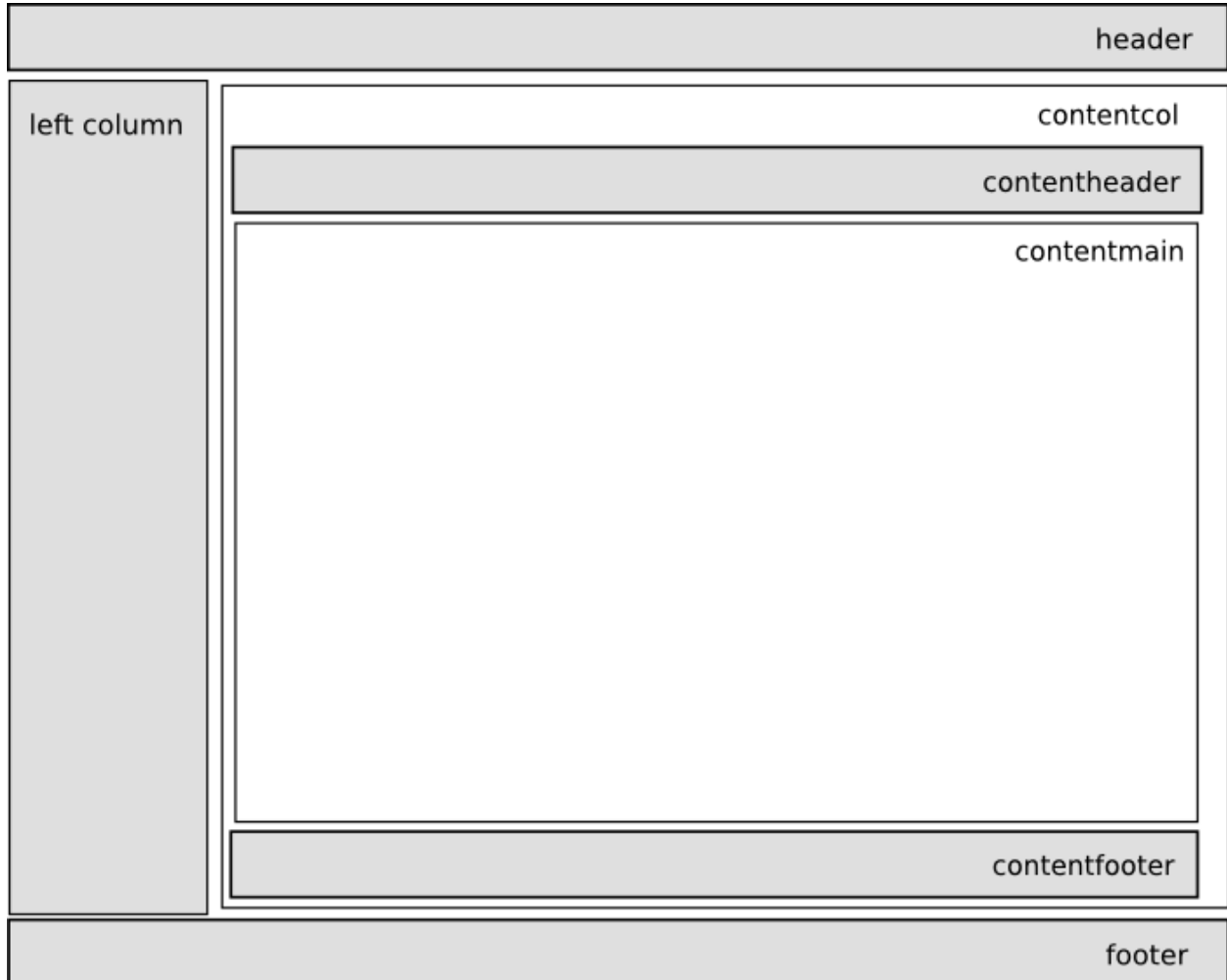
- set up the proper document header and content type
- define the general layout of a document
- invoke adequate views in the various sections of the document

Look at `cubicweb.web.views.basetemplates` and you will find the base templates used to generate (X)HTML for your application. The most important template there is `TheMainTemplate`.

TheMainTemplate

Layout and sections

A page is composed as indicated on the schema below :



The sections dispatches specific views:

- *header*: the rendering of the header is delegated to the *htmlheader* view, whose default implementation can be found in `basetemplates.py` and which does the following things:
 - inject the favicon if there is one
 - inject the global style sheets and javascript resources
 - call and display a link to an rss component if there is one available

it also sets up the page title, and fills the actual *header* section with top-level components, using the *header* view, which:

- tries to display a logo, the name of the application and the *breadcrumbs*
- provides a login status area
- provides a login box (hidden by default)

- *left column*: this is filled with all selectable boxes matching the *left* context (there is also a right column but nowadays it is seldom used due to bad usability)
- *contentcol*: this is the central column; it is filled with:
 - the *rqlinput* view (hidden by default)
 - the *aplmessages* component
 - the *contentheader* view which in turns dispatches all available content navigation components having the *navtop* context (this is used to navigate through entities implementing the IPrevNext interface)
 - the view that was given as input to the template's *call* method, also dealing with pagination concerns
 - the *contentfooter*
- *footer*: adds all footer actions

Note: How and why a view object is given to the main template is explained in the [Publisher](#) chapter.

Configure the main template

You can overload some methods of the `TheMainTemplate`, in order to fulfil your needs. There are also some attributes and methods which can be defined on a view to modify the base template behaviour:

- *paginable*: if the result set is bigger than a configurable size, your result page will be paginated by default. You can set this attribute to *False* to avoid this.
- *binary*: boolean flag telling if the view generates some text or a binary stream. Default to *False*. When view generates text argument given to *self.w* **must be a unicode string**, encoded string otherwise.
- *content_type*, view's content type, default to 'text/xhtml'
- *templatable*, boolean flag telling if the view's content should be returned directly (when *False*) or included in the main template layout (including header, boxes and so on).
- *page_title()*, method that should return a title that will be set as page title in the html headers.
- *html_headers()*, method that should return a list of HTML headers to be included the html headers.

You can also modify certain aspects of the main template of a page when building a url or setting these parameters in the req.form:

- *__notemplate*, if present (whatever the value assigned), only the content view is returned
- *__force_display*, if present and its value is not null, no pagination whatever the number of entities to display (e.g. similar effect as view's *paginable* attribute described above).
- *__method*, if the result set to render contains only one entity and this parameter is set, it refers to a method to call on the entity by passing it the dictionary of the forms parameters, before going the classic way (through step 1 and 2 described just above)
- *vtile*, a title to be set as <h1> of the content

Other templates

There are also the following other standard templates:

- `cubicweb.web.views.basetemplates.LogInTemplate`
- `cubicweb.web.views.basetemplates.LogOutTemplate`
- `cubicweb.web.views.basetemplates.ErrorTemplate` specializes `TheMainTemplate` to do proper end-user output if an error occurs during the computation of `TheMainTemplate` (it is a fallback view).

6.5.3 The Primary View

By default, *CubicWeb* provides a view that fits every available entity type. This is the first view you might be interested in modifying. It is also one of the richest and most complex.

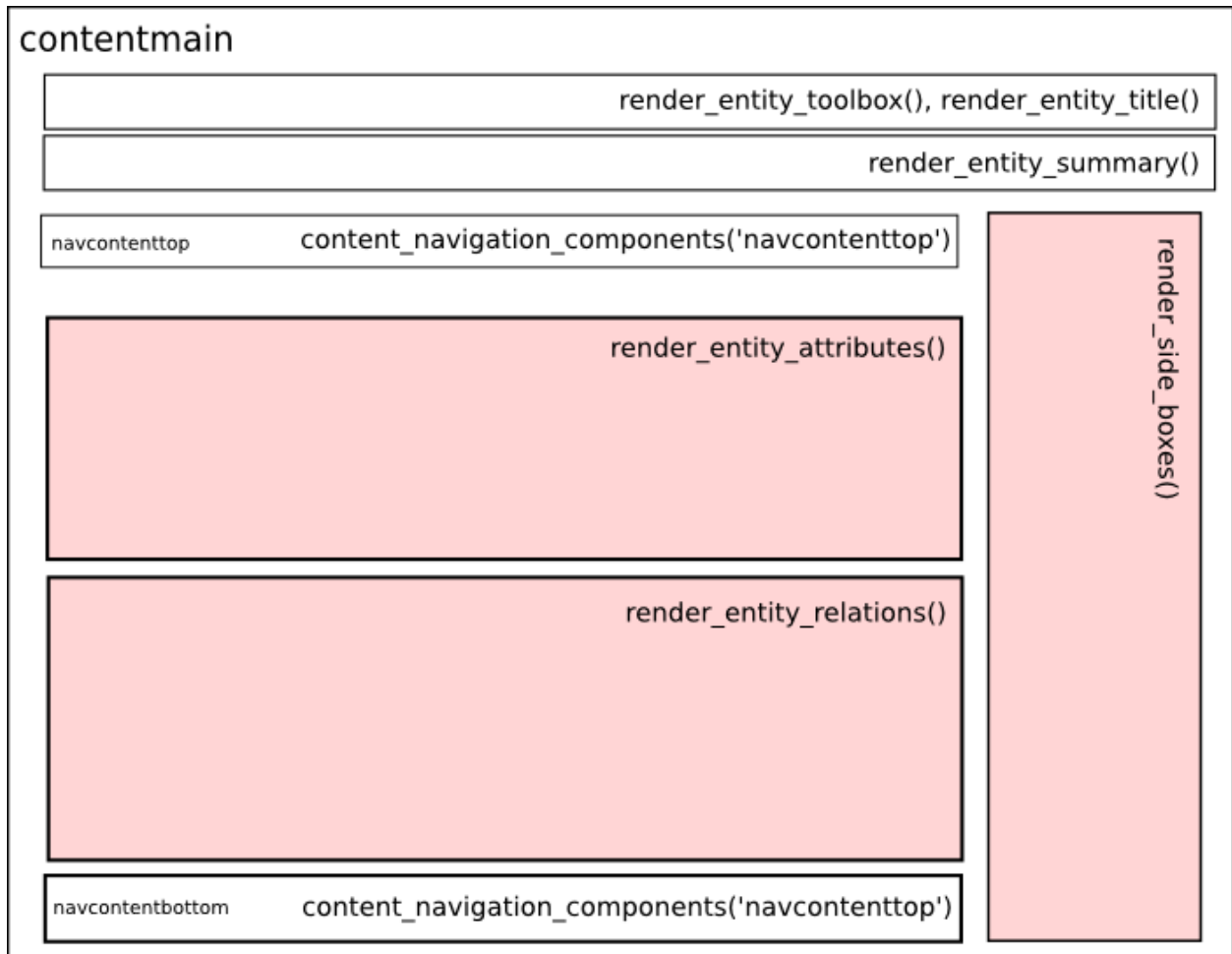
It is automatically selected on a one line result set containing an entity.

It lives in the `cubicweb.web.views.primary` module.

The *primary* view is supposed to render a maximum of informations about the entity.

Layout

The primary view has the following layout.



Primary view configuration

If you want to customize the primary view of an entity, overriding the primary view class may not be necessary. For simple adjustments (attributes or relations display locations and styles), a much simpler way is to use `uicfg`.

Attributes/relations display location

In the primary view, there are three sections where attributes and relations can be displayed (represented in pink in the image above):

- ‘attributes’
- ‘relations’
- ‘sideboxes’

Attributes can only be displayed in the attributes section (default behavior). They can also be hidden. By default, attributes of type *Password* and *Bytes* are hidden.

For instance, to hide the `title` attribute of the `Blog` entity:

```
from cubicweb.web.views import uicfg
uicfg.primaryview_section.tag_attribute(('Blog', 'title'), 'hidden')
```

Relations can be either displayed in one of the three sections or hidden.

For relations, there are two methods:

- `tag_object_of` for modifying the primary view of the object
- `tag_subject_of` for modifying the primary view of the subject

These two methods take two arguments:

- a triplet (subject, relation_name, object), where subject or object can be replaced with '*'
- the section name or hidden

```
pv_section = uicfg.primaryview_section
# hide every relation `entry_of` in the `Blog` primary view
pv_section.tag_object_of('*', 'entry_of', 'Blog'), 'hidden')

# display `entry_of` relations in the `relations`
# section in the `BlogEntry` primary view
pv_section.tag_subject_of(('BlogEntry', 'entry_of', '*'), 'relations')
```

Display content

You can use `primaryview_display_ctrl` to customize the display of attributes or relations. Values of `primaryview_display_ctrl` are dictionaries.

Common keys for attributes and relations are:

- `vid`: specifies the regid of the view for displaying the attribute or the relation.

If vid is not specified, the default value depends on the section:

- attributes section: 'reledit' view
 - relations section: 'autolimited' view
 - sideboxes section: 'sidebox' view
- `order`: int used to control order within a section. When not specified, automatically set according to order in which tags are added.
 - `label`: label for the relations section or side box
 - `showlabel`: boolean telling whether the label is displayed

```
# let us remind the schema of a blog entry
class BlogEntry(EntityType):
    title = String(required=True, fulltextindexed=True, maxsize=256)
    publish_date = Date(default='TODAY')
    content = String(required=True, fulltextindexed=True)
    entry_of = SubjectRelation('Blog', cardinality='?')

# now, we want to show attributes
# with an order different from that in the schema definition
view_ctrl = uicfg.primaryview_display_ctrl
for index, attr in enumerate('title', 'content', 'publish_date'):
    view_ctrl.tag_attribute(('BlogEntry', attr), {'order': index})
```

By default, relations displayed in the ‘relations’ section are being displayed by the ‘autolimited’ view. This view will use comma separated values, or list view and/or limit your rset if there is too much items in it (and generate the “view all” link in this case).

You can control this view by setting the following values in the *primaryview_display_ctrl* relation tag:

- *limit*, maximum number of entities to display. The value of the ‘navigation.related-limit’ cwproperty is used by default (which is 8 by default). If None, no limit.
- *use_list_limit*, number of entities until which they should be display as a list (eg using the ‘list’ view). Below that limit, the ‘csv’ view is used. If None, display using ‘csv’ anyway.
- *subvid*, the subview identifier (eg view that should be used of each item in the list)

Notice you can also use the *filter* key to set up a callback taking the related result set as argument and returning it filtered, to do some arbitrary filtering that can’t be done using rql for instance.

```
pv_section = uicfg.primaryview_section
# in `CWUser` primary view, display `created_by`
# relations in relations section
pv_section.tag_object_of('*', 'created_by', 'CWUser'), 'relations')

# display this relation as a list, sets the label,
# limit the number of results and filters on comments
def filter_comment(rset):
    return rset.filtered_rset(lambda x: x.e_schema == 'Comment')
pv_ctrl = uicfg.primaryview_display_ctrl
pv_ctrl.tag_object_of('*', 'created_by', 'CWUser'),
                    {'vid': 'list', 'label': _('latest comment(s)'),
                     'limit': True,
                     'filter': filter_comment})
```

Warning: with the *primaryview_display_ctrl* rtag, the subject or the object of the relation is ignored for respectively *tag_object_of* or *tag_subject_of*. To avoid warnings during execution, they should be set to `'*'`.

Public API of the PrimaryView class

class cubicweb.web.views.primary.**PrimaryView**(req=None, rset=None, **kwargs)

The basic layout of a primary view is as in the [Layout](#) section. This layout is actually drawn by the *render_entity* method.

The methods you may want to modify while customizing a PrimaryView are:

render_entity_title(entity)

Renders the entity title, by default using entity’s *dc_title()* method.

render_entity_attributes(entity)

Renders all attributes and relations in the ‘attributes’ section.

render_entity_relations(entity)

Renders all relations in the ‘relations’ section.

render_side_boxes(*boxes*)

Renders side boxes on the right side of the content. This will generate a box for each relation in the ‘sidebar’ section, as well as explicit box appobjects selectable in this context.

The placement of relations in the relations section or in side boxes can be controlled through the *Primary view configuration* mechanism.

content_navigation_components(*context*)

This method is applicable only for entity type implementing the interface *IPrevNext*. This interface is for entities which can be linked to a previous and/or next entity. This method will render the navigation links between entities of this type, either at the top or at the bottom of the page given the context (navcontent{top|bottom}).

Also, please note that by setting the following attributes in your subclass, you can already customize some of the rendering:

show_attr_label Renders the attribute label next to the attribute value if set to *True*. Otherwise, does only display the attribute value.

show_rel_label Renders the relation label next to the relation value if set to *True*. Otherwise, does only display the relation value.

main_related_section Renders the relations of the entity if set to *True*.

A good practice is for you to identify the content of your entity type for which the default rendering does not answer your need so that you can focus on the specific method (from the list above) that needs to be modified. We do not advise you to overwrite `render_entity` unless you want a completely different layout.

Views that may be used to display an entity’s attribute or relation

You may easily the display of an attribute or relation by simply configuring the view using one of *primaryview_display_ctrl* or *reledit_ctrl* to use one of the views described below. For instance:

```
primaryview_display_ctrl.tag_attribute(('Foo', 'bar'), {'vid': 'attribute'})
```

```
class cubicweb.web.views.primary.AttributeView(req=None, rset=None, **kwargs)
```

```
    __regid__ attribute
```

This view is generally used to disable the *reledit* feature. It works on both relations and attributes.

```
class cubicweb.web.views.primary.URLAttributeView(req=None, rset=None, **kwargs)
```

```
    __regid__ urlattr
```

This view will wrap an attribute value (hence expect a string) into an ‘<a>’ HTML tag to display a clickable link.

```
class cubicweb.web.views.primary.VerbatimAttributeView(req=None, rset=None, **kwargs)
```

```
    __regid__ verbatimattr
```

This view will wrap an attribute value into an ‘<pre>’ HTML tag to display arbitrary text where EOL will be respected. It usually make sense for attributes whose value is a multi-lines string where new lines matters.

Example of customization and creation

We'll show you now an example of a primary view and how to customize it.

If you want to change the way a `BlogEntry` is displayed, just override the method `cell_call()` of the view `primary` in `BlogDemo/views.py`.

```
from cubicweb.predicates import is_instance
from cubicweb.web.views.primary import Primaryview

class BlogEntryPrimaryView(PrimaryView):
    __select__ = PrimaryView.__select__ & is_instance('BlogEntry')

    def render_entity_attributes(self, entity):
        self.w(u'<p>published on %s</p>' %
              entity.publish_date.strftime('%Y-%m-%d'))
        super(BlogEntryPrimaryView, self).render_entity_attributes(entity)
```

The above source code defines a new primary view for `BlogEntry`. The `__reid__` class attribute is not repeated there since it is inherited through the `primary.PrimaryView` class.

The selector for this view chains the selector of the inherited class with its own specific criterion.

The view method `self.w()` is used to output data. Here *lines 08-09* output HTML for the publication date of the entry.



Let us now improve the primary view of a blog

```
from logilab.mtconverter import xml_escape
from cubicweb.predicates import is_instance, one_line_rset
from cubicweb.web.views.primary import Primaryview

class BlogPrimaryView(PrimaryView):
    __regid__ = 'primary'
    __select__ = PrimaryView.__select__ & is_instance('Blog')
    rql = 'Any BE ORDERBY D DESC WHERE BE entry_of B, BE publish_date D, B eid %(b)s'

    def render_entity_relations(self, entity):
        rset = self._cw.execute(self.rql, {'b' : entity.eid})
        for entry in rset.entities():
            self.w(u'<p>%s</p>' % entry.view('inblogcontext'))
```

(continues on next page)

(continued from previous page)

```
class BlogEntryInBlogView(EntityView):
    __regid__ = 'inblogcontext'
    __select__ = is_instance('BlogEntry')

    def cell_call(self, row, col):
        entity = self.cw_rset.get_entity(row, col)
        self.w(u'<a href="%s" title="%s">%s</a>' %
              entity.absolute_url(),
              xml_escape(entity.content[:50]),
              xml_escape(entity.description))
```

This happens in two places. First we override the `render_entity_relations` method of a Blog's primary view. Here we want to display our blog entries in a custom way.

At *line 10*, a simple request is made to build a result set with all the entities linked to the current `Blog` entity by the relationship `entry_of`. The part of the framework handling the request knows about the schema and infers that such entities have to be of the `BlogEntry` kind and retrieves them (in the prescribed `publish_date` order).

The request returns a selection of data called a result set. Result set objects have an `.entities()` method returning a generator on requested entities (going transparently through the *ORM* layer).

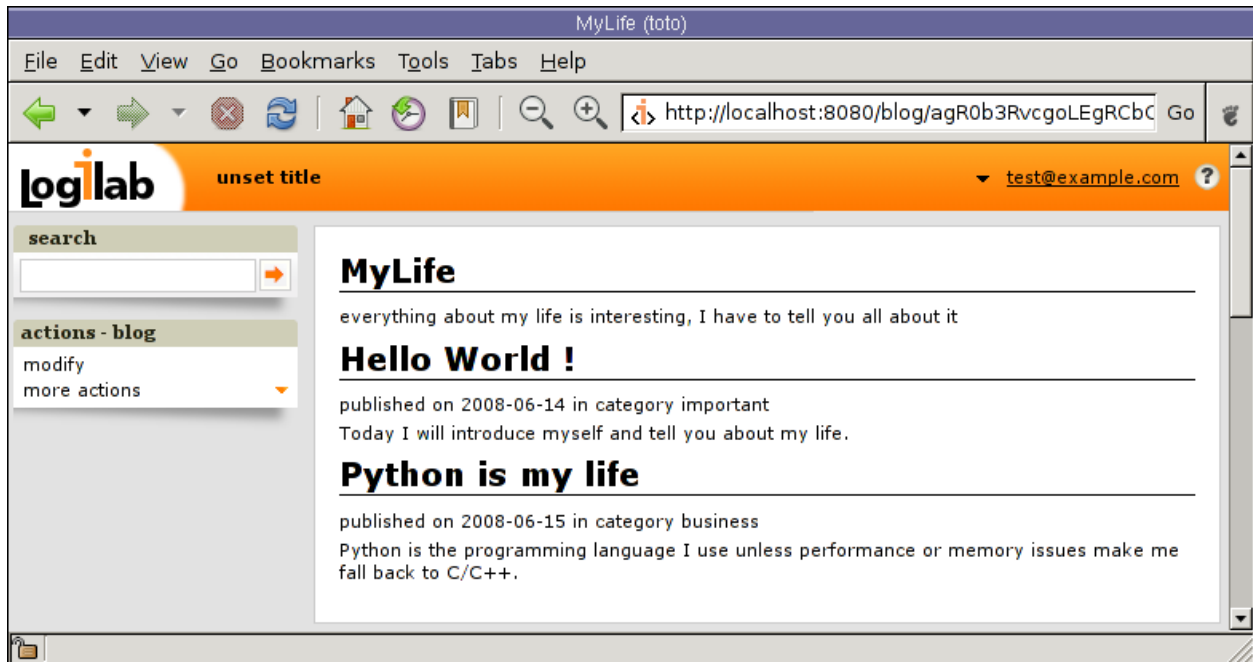
At *line 13* the view 'inblogcontext' is applied to each blog entry to output HTML. (Note that the 'inblogcontext' view is not defined whatsoever in *CubicWeb*. You are absolutely free to define whole view families.) We just arrange to wrap each blogentry output in a 'p' html element.

Next, we define the 'inblogcontext' view. This is NOT a primary view, with its well-defined sections (title, metadata, attributes, relations/boxes). All a basic view has to define is `cell_call`.

Since views are applied to result sets which can be tables of data, we have to recover the entity from its (row,col)-coordinates (*line 20*). Then we can spit some HTML.

Warning: Be careful: all strings manipulated in *CubicWeb* are actually unicode strings. While web browsers are usually tolerant to incoherent encodings they are being served, we should not abuse it. Hence we have to properly escape our data. The `xml_escape()` function has to be used to safely fill (X)HTML elements from Python unicode strings.

Assuming we added entries to the blog titled *MyLife*, displaying it now allows to read its description and all its entries.



6.5.4 The “Click and Edit” (also *reledit*) View

The principal way to update data through the Web UI is through the *modify* action on entities, which brings a full form. This is described in the *HTML form construction* chapter.

There is however another way to perform piecewise edition of entities and relations, using a specific *reledit* (for *relation edition*) view from the `cubicweb.web.views.reledit` module.

This is typically applied from the default Primary View (see *The Primary View*) on the attributes and relation section. It makes small editions more convenient.

Of course, this can be used customely in any other view. Here come some explanation about its capabilities and instructions on the way to use it.

Using *reledit*

Let’s start again with a simple example:

```
class Company(EntityType):
    name = String(required=True, unique=True)
    boss = SubjectRelation('Person', cardinality='1*')
    status = SubjectRelation('File', cardinality='?* ', composite='subject')
```

In some view code we might want to show these attributes/relations and allow the user to edit each of them in turn without having to leave the current page. We would write code as below:

```
company.view('reledit', rtype='name', default_value='<name>') # editable name attribute
company.view('reledit', rtype='boss') # editable boss relation
company.view('reledit', rtype='status') # editable attribute-like relation
```

If one wanted to edit the company from a boss’s point of view, one would have to indicate the proper relation’s role. By default the role is *subject*.

```
person.view('reledit', rtype='boss', role='object')
```

Each of these will provide with a different editing widget. The *name* attribute will obviously get a text input field. The *boss* relation will be edited through a selection box, allowing to pick another *Person* as boss. The *status* relation, given that it defines Company as a composite entity with one file inside, will provide additional actions

- to *add* a *File* when there is one
- to *delete* the *File* (if the cardinality allows it)

Moreover, editing the relation or using the *add* action leads to an embedded edition/creation form allowing edition of the target entity (which is *File* in our example) instead of merely allowing to choose amongst existing files.

The *reledit_ctrl* rtag

The behaviour of reedited attributes/relations can be finely controlled using the *reledit_ctrl* rtag, defined in [cubicweb.web.views.uicfg](#).

This rtag provides four control variables:

- **default_value: alternative default value** The default value is what is shown when there is no value.
- **reload: boolean, eid (to reload to) or function taking subject** and returning bool/eid This is useful when editing a relation (or attribute) that impacts the url or another parts of the current displayed page. Defaults to false.
- **rvid: alternative view id (as str) for relation or composite** edition Default is 'incontext' or 'csv' depending on the cardinality. They can also be statically changed by subclassing ClickAndEditFormView and redefining *_one_rvid* (resp. *_many_rvid*).
- **edit_target: 'rtype' (to edit the relation) or 'related' (to edit the related entity)** This controls whether to edit the relation or the target entity of the relation. Currently only one-to-one relations support target entity edition. By default, the 'related' option is taken whenever the relation is composite and one-to-one.

Let's see how to use these controls.

```
from logilab.mtconverter import xml_escape
from cubicweb.web.views.uicfg import reledit_ctrl
reledit_ctrl.tag_attribute(('Company', 'name'),
                          {'reload': lambda x:x.eid,
                           'default_value': xml_escape(u'<logilab tastes better>')})
reledit_ctrl.tag_object_of('*', 'boss', 'Person'), {'edit_target': 'related'})
```

The *default_value* needs to be an xml escaped unicode string.

The *edit_target* tag on the *boss* relation being set to *related* will ensure edition of the *Person* entity instead (using a standard automatic form) of the association of Company and Person.

Finally, the *reload* key accepts either a boolean, an eid or a unicode string representing a url. If an eid is provided, it will be internally transformed into a url. The eid/url case helps when one needs to reload and the current url is inappropriate. A common case is edition of a key attribute, which is part of the current url. If one user changed the Company's name from *lozilab* to *logilab*, reloading on <http://myapp/company/lozilab> would fail. Providing the entity's eid, then, forces to reload on something like <http://myapp/company/42>, which always work.

Disable *reledit*

By default, *reledit* is available on attributes and relations displayed in the ‘attribute’ section of the default primary view. If you want to disable it for some attribute or relation, you have use *uicfg*:

```
from cubicweb.web.views.uicfg import primaryview_display_ctrl as _pvdc
_pvdc.tag_attribute(('Company', 'name'), {'vid': 'incontext'})
```

To deactivate it everywhere it’s used automatically, you may use the code snippet below somewhere in your cube’s views:

```
from cubicweb.web.views import reledit

class DeactivatedAutoClickAndEditFormView(reledit.AutoClickAndEditFormView):
    def _should_edit_attribute(self, rschema):
        return False

    def _should_edit_attribute(self, rschema, role):
        return False

def registration_callback(vreg):
    vreg.register_and_replace(DeactivatedAutoClickAndEditFormView,
                             reledit.AutoClickAndEditFormView)
```

6.5.5 Base views

CubicWeb provides a lot of standard views, that can be found in `cubicweb.web.views` sub-modules.

A certain number of views are used to build the web interface, which apply to one or more entities. As other appobjects, their identifier is what distinguish them from each others. The most generic ones, found in `cubicweb.web.views.baseviews`, are described below.

You’ll probably want to customize one or more of the described views which are default, generic, implementations.

HTML views

Special views

```
class cubicweb.web.views.baseviews.NullView(req=None, rset=None, **kwargs)
```

```
    __regid__ null
```

This view is the default view used when nothing needs to be rendered. It is always applicable and is usually used as fallback view when calling `_cw.view()` to display nothing if the result set is empty.

```
class cubicweb.web.views.baseviews.NoResultView(req=None, rset=None, **kwargs)
```

```
    __regid__ noresult
```

This view is the default view to be used when no result has been found (i.e. empty result set).

It’s usually used as fallback view when calling `_cw.view()` to display “no results” if the result set is empty.

```
class cubicweb.web.views.baseviews.FinalView(req=None, rset=None, **kwargs)
```

```
    __regid__ final
```

Display the value of a result set cell with minimal transformations (i.e. you'll get a number for entities). It is applicable on any result set, though usually dedicated for cells containing an attribute's value.

Base entity views

```
class cubicweb.web.views.baseviews.InContextView(req=None, rset=None, **kwargs)
```

```
    __regid__ incontext
```

This view is used when the entity should be considered as displayed in its context. By default it produces the result of `entity.dc_title()` wrapped in a link leading to the primary view of the entity.

```
class cubicweb.web.views.baseviews.OutOfContextView(req=None, rset=None, **kwargs)
```

```
    __regid__ outofcontext
```

This view is used when the entity should be considered as displayed out of its context. By default it produces the result of `entity.dc_long_title()` wrapped in a link leading to the primary view of the entity.

```
class cubicweb.web.views.baseviews.OneLineView(req=None, rset=None, **kwargs)
```

```
    __regid__ oneline
```

This view is used when we can't tell if the entity should be considered as displayed in or out of context. By default it produces the result of the *text* view in a link leading to the primary view of the entity.

Those are used to display a link to an entity, whose label depends on the entity having to be displayed in or out of context (of another entity): some entities make sense in the context of another entity. For instance, the *Version* of a *Project* in forge. So one may expect that 'incontext' will be called when display a version from within the context of a project, while 'outofcontext' will be called in other cases. In our example, the 'incontext' view of the version would be something like '0.1.2', while the 'outofcontext' view would include the project name, e.g. 'baz 0.1.2' (since only a version number without the associated project doesn't make sense if you don't know yet that you're talking about the famous 'baz' project. *CubicWeb* tries to make guess and call 'incontext'/'outofcontext' nicely. When it can't know, the 'oneline' view should be used.

List entity views

```
class cubicweb.web.views.baseviews.ListView(req=None, rset=None, **kwargs)
```

```
    __regid__ list
```

This view displays a list of entities by creating a HTML list (``) and call the view *listitem* for each entity of the result set. The 'list' view will generate HTML like:

```
<ul class="section">
  <li>"result of 'subvid' view for a row</li>
  ...
</ul>
```

If you wish to use a different view for each entity, either subclass and change the `item_vid` class attribute or specify a *subvid* argument when calling this view.

```
class cubicweb.web.views.baseviews.SimpleListView(req=None, rset=None, **kwargs)
```

```
    __regid__ simplelist
```

Similar to :class:`~cubicweb.web.views.baseviews.ListView` but using '`<div>`' instead of '``'. It rely on '`<div>`' behaviour to separate items. HTML will look like

```
<div class="section">"result of 'subvid' view for a row</div>
...
```

It relies on base [View](#) class implementation of the `call()` method to insert those `<div>`.

```
class cubicweb.web.views.baseviews.SameETypeListView(req=None, rset=None, **kwargs)
```

```
    __regid__ sameetypelist
```

This view displays a list of entities of the same type, in HTML section (`<div>`) and call the view *sameetypelist-item* for each entity of the result set. It's designed to get a more adapted global list when displayed entities are all of the same type (for instance, display gallery if there are only images entities).

```
class cubicweb.web.views.baseviews.CSVView(req=None, rset=None, **kwargs)
```

```
    __regid__ csv
```

This view displays each entity in a coma separated list. It is NOT related to the well-known text file format.

Those list views can be given a `'subvid'` arguments, telling the view to use of each item in the list. When not specified, the value of the `'redirect_vid'` attribute of `ListItemView` (for `'listview'`) or of [SimpleListView](#) will be used. This default to `'outofcontext'` for `'list'` / `'incontext'` for `'simplelist'`

Text entity views

Basic HTML view have some variants to be used when generating raw text, not HTML (for notifications for instance). Also, as explained above, some of the HTML views use those text views as a basis.

```
class cubicweb.web.views.baseviews.TextView(req=None, rset=None, **kwargs)
```

```
    __regid__ text
```

This is the simplest text view for an entity. By default it returns the result of the entity's `dc_title()` method, which is cut to fit the *navigation.short-line-size* property if necessary.

```
class cubicweb.web.views.baseviews.InContextTextView(req=None, rset=None, **kwargs)
```

```
    __regid__ textincontext
```

Similar to the *text* view, but called when an entity is considered in context (see description of incontext HTML view for more information on this). By default it displays what's returned by the `dc_title()` method of the entity.

```
class cubicweb.web.views.baseviews.OutOfContextView(req=None, rset=None, **kwargs)
```

```
    __regid__ outofcontext
```

This view is used when the entity should be considered as displayed out of its context. By default it produces the result of `entity.dc_long_title()` wrapped in a link leading to the primary view of the entity.

You will also find modules providing some specific services:

This module provides some generic components to navigate in the web application.

Pagination

Several implementations for large result set pagination are provided:

class cubicweb.web.views.navigation.**PageNavigation**(*req, rset, **kwargs*)

The default pagination component: display link to pages where each pages is identified by the item number of its first and last elements.

class cubicweb.web.views.navigation.**PageNavigationSelect**(*req, rset, **kwargs*)

This pagination component displays a result-set by page as [PageNavigation](#) but in a <select>, which is better when there are a lot of results.

By default it will be selected when there are more than 4 pages to be displayed.

class cubicweb.web.views.navigation.**SortedNavigation**(*req, rset, **kwargs*)

This pagination component will be selected by default if there are less than 4 pages and if the result set is sorted.

Displayed links to navigate accross pages of a result set are done according to the first variable on which the sort is done, and looks like:

```
[ana - cro] | [cro - ghe] | ... | [tim - zou]
```

You may want to override this component to customize display in some cases.

sort_on()

Return entity column number / attr name to use for nice display by inspecting the rset'syntax tree.

display_func(*rset, col, attrname*)

Return a function that will be called with a row number as argument and should return a string to use as link for it.

format_link_content(*startstr, stopstr*)

Return text for a page link, where *startstr* and *stopstr* are the text for the lower/upper boundaries of the page.

By default text are stripped down to `nb_chars` characters.

write_links(*basepath, params, blocklist*)

Return HTML for the whole navigation: *blocklist* is a list of HTML snippets for each page, *basepath* and *params* will be necessary to build previous/next links.

Below an example from the tracker cube:

```
class TicketsNavigation(navigation.SortedNavigation):
    __select__ = (navigation.SortedNavigation.__select__
                  & ~paginated_rset(4) & is_instance('Ticket'))
    def sort_on(self):
        col, attrname = super(TicketsNavigation, self).sort_on()
        if col == 6:
            # sort on state, we don't want that
            return None, None
        return col, attrname
```

The idea is that in trackers'ticket tables, result set is first ordered on ticket's state while this doesn't make any sense in the navigation. So we override `sort_on()` so that if we detect such sorting, we disable the feature to go back to item number in the pagination.

Also notice the `~paginated_rset(4)` in the selector so that if there are more than 4 pages to display, [PageNavigationSelect](#) will still be selected.

Pagination will appear when needed according to the *page-size* ui property.

This module monkey-patch the `paginate()` function to the base View class, so that you can ask pagination explicitly on every result-set based views.

```
cubicweb.web.views.navigation.paginate(view, show_all_option=True, w=None, page_size=None,
                                       rset=None)
```

paginate results if the view is paginable

Previous / next navigation

An adapter and its related component for the somewhat usual “previous / next” navigation are provided.

```
class cubicweb.web.views.navigation.IPrevNextAdapter(_cw, **kwargs)
    Interface for entities which can be linked to a previous and/or next entity

    next_entity()
        return the ‘next’ entity

    previous_entity()
        return the ‘previous’ entity

class cubicweb.web.views.navigation.NextPrevNavigationComponent(*args, **kwargs)
    Entities adaptable to the ‘IPrevNext’ should have this component automatically displayed. You may
    want to override this component to have a different look and feel.
```

6.5.6 Startup views

Startup views are views requiring no context, from which you usually start browsing (for instance the index page). The usual selectors are *none_rset* or *yes*.

You’ll find here a description of startup views provided by the framework.

This module contains the default index page and management view.

```
class cubicweb.web.views.startup.IndexView(req=None, rset=None, **kwargs)
```

__regid__ *index*

The default index view, that you’ll get when accessing your site’s root url. It’s by default identical to the *ManageView*, but you’ll usually want to customize this one.

```
class cubicweb.web.views.startup.ManageView(req=None, rset=None, **kwargs)
```

__regid__ *manage*

The manage view, display some information about what’s contained by your site and provides access to administration stuff such as user and groups management.

Regarding the section displaying link to entity type, notice by default it won’t display entity types which are related to another one using a mandatory (cardinality == 1) composite relation.

You can still configure that behaviour manually using the *indexview_etype_section* as explained in `cubicweb.web.uicfg`.

Other startup views:

schema A view dedicated to the display of the schema of the instance

6.5.7 Boxes

(cubicweb.web.views_boxes)

sidebox This view displays usually a side box of some related entities in a primary view.

The action box

The `add_related` is an automatic menu in the action box that allows to create an entity automatically related to the initial entity (context in which the box is displayed). By default, the links generated in this box are computed from the schema properties of the displayed entity, but it is possible to explicitly specify them thanks to the `cubicweb.web.views.uicfg.rmode` relation tag:

- *link*, indicates that a relation is in general created pointing to an existing entity and that we should not to display a link for this relation
- *create*, indicates that a relation is in general created pointing to new entities and that we should display a link to create a new entity and link to it automatically

If necessary, it is possible to overwrite the method `relation_mode(rtype, targettype, x='subject')` to dynamically compute a relation creation category.

Please note that if at least one action belongs to the *addrelated* category, the automatic behavior is deactivated in favor of an explicit behavior (e.g. display of *addrelated* category actions only).

6.5.8 Table views

This module contains table views, with the following features that may be provided (depending on the used implementation):

- facets filtering
- pagination
- actions menu
- properly sortable content
- odd/row/hover line styles

The three main implementation are described below. Each implementation is suitable for a particular case, but they each attempt to display tables that looks similar.

class cubicweb.web.views.tableview.**RsetTableView**(req=None, rset=None, **kwargs)

This table view accepts any non-empty rset. It uses introspection on the result set to compute column names and the proper way to display the cells.

It is highly configurable and accepts a wealth of options, but take care to check what you're trying to achieve wouldn't be a job for the [EntityTableView](#). Basically the question is: does this view should be tied to the result set query's shape or no? If yes, than you're fine. If no, you should take a look at the other table implementation.

The following class attributes may be used to control the table:

- *finalvid*, a view identifier that should be called on final entities (e.g. attribute values). Default to 'final'.
- *nonfinalvid*, a view identifier that should be called on entities. Default to 'incontext'.
- *displaycols*, if not *None*, should be a list of rset's columns to be displayed.
- *headers*, if not *None*, should be a list of headers for the table's columns. *None* values in the list will be replaced by computed column names.

- *cellvids*, if not *None*, should be a dictionary with table column index as key and a view identifier as value, telling the view that should be used in the given column.

Notice *displaycols*, *headers* and *cellvids* may be specified at selection time but then the table won't have pagination and shouldn't be configured to display the facets filter nor actions (as they wouldn't behave as expected).

This table class use the [RsetTableColRenderer](#) as default column renderer.

class RsetTableColRenderer(*cellvid*, ***kwargs*)

Default renderer for [RsetTableView](#).

build_column_renderers()

Return a list of column renderers, one for each column to be rendered. Prototype of a column renderer is described below:

class AbstractColumnRenderer(*header=None*, *addcount=False*, *trheader=True*,
escapeheader=True, *sortable=True*)

Abstract base class for column renderer. Interface of a column renderer follows:

AbstractColumnRenderer.bind(*view*, *colid*)

Bind the column renderer to its view. This is where *_cw*, *view*, *colid* are set and the method to override if you want to add more view/request depending attributes on your column render.

AbstractColumnRenderer.render_header(*w*)

Write label for the specified column by calling *w()*.

AbstractColumnRenderer.render_cell(*w*, *rownum*)

Write value for the specified cell by calling *w()*.

Parameters *rownum* – the row number in the table

AbstractColumnRenderer.sortvalue(*_rownum*)

Return typed value to be used for sorting on the specified column.

Parameters *rownum* – the row number in the table

Attributes on this base class are:

Attr *header*, the column header. If *None*, default to *_(colid)*

Attr *addcount*, if *True*, add the table size in parenthesis beside the header

Attr *trheader*, should the header be translated

Attr *escapeheader*, should the header be *xml_escaped*

Attr *sortable*, tell if the column is sortable

Attr *view*, the table view

Attr *_cw*, the request object

Attr *colid*, the column identifier

Attr *attributes*, dictionary of attributes to put on the HTML tag when the cell is rendered

call(*headers=None*, *displaycols=None*, *cellvids=None*, *paginate=None*)

the view is called for an entire result set, by default loop other rows of the result set and call the same view on the particular row

Views applicable on *None* result sets have to override this method

default_column_renderer_class

alias of `cubicweb.web.views.tableview.RsetTableColRenderer`

linkable()

return True if the view may be linked in a menu
by default views without title are not meant to be displayed

main_var_index()

returns the index of the first non-attribute variable among the RQL selected variables

property table_size

return the number of rows (header excluded) to be displayed

class cubicweb.web.views.tableview.**EntityTableView**(*req=None, rset=None, **kwargs*)

This abstract table view is designed to be used with an `is_instance()` or `adaptable` predicate, hence doesn't depend the result set shape as the [RsetTableView](#) does.

It will display columns that should be defined using the `columns` class attribute containing a list of column ids. By default, each column is rendered by [EntityTableColRenderer](#) which consider that the column id is an attribute of the table's main entity (ie the one for which the view is selected).

You may wish to specify [MainEntityColRenderer](#) or [RelatedEntityColRenderer](#) renderer for a column in the `column_renderers` dictionary.

class **EntityTableColRenderer**(*renderfunc=None, sortfunc=None, sortable=None, **kwargs*)

Default column renderer for [EntityTableView](#).

You may use the `entity()` method to retrieve the main entity for a given row number.

EntityTableColRenderer.entity(*rownum*)

Convenience method returning the table's main entity.

EntityTableColRenderer.render_entity(*w, entity*)

Sort value if *renderfunc* nor *sortfunc* specified at initialization.

This default implementation consider column id is an entity attribute and print its value.

EntityTableColRenderer.entity_sortvalue(*entity*)

Cell rendering implementation if *renderfunc* nor *sortfunc* specified at initialization.

This default implementation consider column id is an entity attribute and return its sort value by calling *entity.sortvalue(colid)*.

class **MainEntityColRenderer**(*vid='incontext', addcount=True, **kwargs*)

Renderer to be used for the column displaying the 'main entity' of a [EntityTableView](#).

By default display it using the 'incontext' view. You may specify another view identifier using the *vid* argument.

If header not specified, it would be built using entity types in the main column.

class **RelatedEntityColRenderer**(*getrelated, addcount=False, **kwargs*)

Renderer to be used for column displaying an entity related the 'main entity' of a [EntityTableView](#).

By default display it using the 'incontext' view. You may specify another view identifier using the *vid* argument.

If header not specified, it would be built by translating the column id.

class **RelationColRenderer**(*role='subject', vid='csv', subvid=None, fallbackvid='empty-cell', is_rtype_view=False, **kwargs*)

Renderer to be used for column displaying a list of entities related the 'main entity' of a

EntityTableView. By default, the main entity is considered as the subject of the relation but you may specify otherwise using the *role* argument.

By default display the related rset using the ‘csv’ view, using ‘outofcontext’ sub-view for each entity. You may specify another view identifier using respectively the *vid* and *subvid* arguments.

If you specify a ‘rtype view’, such as ‘reledit’, you should add a *is_rtype_view=True* parameter.

If header not specified, it would be built by translating the column id, properly considering role.

build_column_renderers()

Return a list of column renderers, one for each column to be rendered. Prototype of a column renderer is described below:

```
class AbstractColumnRenderer(header=None, addcount=False, trheader=True,
                               escapeheader=True, sortable=True)
```

Abstract base class for column renderer. Interface of a column renderer follows:

AbstractColumnRenderer.bind(view, colid)

Bind the column renderer to its view. This is where *_cw*, *view*, *colid* are set and the method to override if you want to add more view/request depending attributes on your column render.

AbstractColumnRenderer.render_header(w)

Write label for the specified column by calling w().

AbstractColumnRenderer.render_cell(w, rownum)

Write value for the specified cell by calling w().

Parameters *rownum* – the row number in the table

AbstractColumnRenderer.sortvalue(_rownum)

Return typed value to be used for sorting on the specified column.

Parameters *rownum* – the row number in the table

Attributes on this base class are:

Attr *header*, the column header. If None, default to *_(colid)*

Attr *addcount*, if True, add the table size in parenthesis beside the header

Attr *trheader*, should the header be translated

Attr *escapeheader*, should the header be xml_escaped

Attr *sortable*, tell if the column is sortable

Attr *view*, the table view

Attr *_cw*, the request object

Attr *colid*, the column identifier

Attr *attributes*, dictionary of attributes to put on the HTML tag when the cell is rendered

call(columns=None, **kwargs)

the view is called for an entire result set, by default loop other rows of the result set and call the same view on the particular row

Views applicable on None result sets have to override this method

default_column_renderer_class

alias of `cubicweb.web.views.tableview.EntityTableColRenderer`

entity(*rownum*)

Return the table's main entity

property table_size

Return the number of rows (header excluded) to be displayed.

By default return the number of rows in the view's result set. If your table isn't result set based, override this method.

class cubicweb.web.views.pyviews.**PyValTableView**(*req, pyvalue, headers=None, cssclass=None, header_column_idx=None, **kwargs*)

This table view is designed to be used a list of list of unicode values given as a mandatory *pyvalue* argument. Take care, content is NOT xml-escaped.

It's configured through the following selection arguments.

If *headers* is specified, it is expected to be a list of headers to be inserted as first row (in <thead>).

header_column_idx may be used to specify a column index or a set of column indices where values should be inserted inside <th> tag instead of <td>.

cssclass is the CSS class used on the <table> tag, and default to 'listing' (so that the table will look similar to those generated by the table view).

build_column_renderers()

Return a list of column renderers, one for each column to be rendered. Prototype of a column renderer is described below:

class **AbstractColumnRenderer**(*header=None, addcount=False, trheader=True, escapeheader=True, sortable=True*)

Abstract base class for column renderer. Interface of a column renderer follows:

AbstractColumnRenderer.bind(*view, colid*)

Bind the column renderer to its view. This is where *_cw*, *view*, *colid* are set and the method to override if you want to add more view/request depending attributes on your column render.

AbstractColumnRenderer.render_header(*w*)

Write label for the specified column by calling *w()*.

AbstractColumnRenderer.render_cell(*w, rownum*)

Write value for the specified cell by calling *w()*.

Parameters *rownum* – the row number in the table

AbstractColumnRenderer.sortvalue(*_rownum*)

Return typed value to be used for sorting on the specified column.

Parameters *rownum* – the row number in the table

Attributes on this base class are:

Attr *header*, the column header. If None, default to *_(colid)*

Attr *addcount*, if True, add the table size in parenthesis beside the header

Attr *trheader*, should the header be translated

Attr *escapeheader*, should the header be xml_escaped

Attr *sortable*, tell if the column is sortable

Attr *view*, the table view

Attr *_cw*, the request object

Attr *colid*, the column identifier

Attr *attributes*, dictionary of attributes to put on the HTML tag when the cell is rendered

default_column_renderer_class

alias of `cubicweb.web.views.pyviews.PyValTableColRenderer`

domid = None

property has_headers

`bool(x) -> bool`

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property layout_args

`dict()` -> new empty dictionary `dict(mapping)` -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: `d = {}` for k, v in iterable:

`d[k] = v`

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list.

For example: `dict(one=1, two=2)`

table_actions()

Return a list of actions ([Link](#)) that match the view's result set, and return those in the 'mainactions' category.

property table_size

return the number of rows (header excluded) to be displayed

All those classes are rendered using a *layout*:

class `cubicweb.web.views.tableview.TableLayout`(*req*, *view*, ***kwargs*)

The default layout for table. When *render* is called, this will use the API described on [TableMixIn](#) to feed the generated table.

This layout behaviour may be customized using the following attributes / selection arguments:

- *cssclass*, a string that should be used as HTML class attribute. Default to "listing".
- *needs_css*, the CSS files that should be used together with this table. Default to ('cubicweb.tablesorter.css', 'cubicweb.tableview.css').
- *needs_js*, the Javascript files that should be used together with this table. Default to ('jquery.tablesorter.js',)
- *display_filter*, tells if the facets filter should be displayed when possible. Allowed values are: - *None*, don't display it - 'top', display it above the table - 'bottom', display it below the table
- *display_actions*, tells if a menu for available actions should be displayed when possible (see two following options). Allowed values are: - *None*, don't display it - 'top', display it above the table - 'bottom', display it below the table
- *hide_filter*, when true (the default), facets filter will be hidden by default, with an action in the actions menu allowing to show / hide it.
- *show_all_option*, when true, a *show all results* link will be displayed below the navigation component.
- *add_view_actions*, when true, actions returned by `view.table_actions()` will be included in the actions menu.

- *header_column_idx*, if not *None*, should be a column index or a set of column index where `<th>` tags should be generated instead of `<td>`

initial_load

<wrapped by the cachedproperty decorator> We detect a bit heuristically if we are built for the first time or

from subsequent calls by the form filter or by the pagination hooks.

render(*w*, ***kwargs*)

called to render a view object for a result set.

This method is dispatched to an actual method selected according to optional row and col parameters, which are locating a particular row or cell in the result set:

- if row is specified, *cell_call* is called
- if none of them is supplied, the view is considered to apply on the whole result set (which may be *None* in this case), *call* is called

There is by default only one table layout, using the ‘table_layout’ identifier, that is referenced by table views `cubicweb.web.views.tableview.TableMixIn.layout_id`. If you want to customize the look and feel of your table, you can either replace the default one by yours, having multiple variants with proper selectors, or change the *layout_id* identifier of your table to use your table specific implementation.

Notice you can give options to the layout using a *layout_args* dictionary on your class.

If you still can’t find a view that suit your needs, you should take a look at the class below that is the common abstract base class for the three views defined above and implement your own class.

class `cubicweb.web.views.tableview.TableMixIn`

Abstract mix-in class for layout based tables.

This default implementation’s call method simply delegate to meth:*layout_render* that will select the renderer whose identifier is given by the *layout_id* attribute.

Then it provides some default implementation for various parts of the API used by that layout.

Abstract method you will have to override is:

build_column_renderers()

Return a list of column renderers, one for each column to be rendered. Prototype of a column renderer is described below:

```
class AbstractColumnRenderer(header=None, addcount=False, trheader=True,  
                             escapeheader=True, sortable=True)
```

Abstract base class for column renderer. Interface of a column renderer follows:

`AbstractColumnRenderer.bind`(*view, colid*)

Bind the column renderer to its view. This is where *_cw*, *view*, *colid* are set and the method to override if you want to add more view/request depending attributes on your column render.

`AbstractColumnRenderer.render_header`(*w*)

Write label for the specified column by calling *w()*.

`AbstractColumnRenderer.render_cell`(*w, rownum*)

Write value for the specified cell by calling *w()*.

Parameters *rownum* – the row number in the table

AbstractColumnRenderer.sortvalue(_rownum)

Return typed value to be used for sorting on the specified column.

Parameters **rownum** – the row number in the table

Attributes on this base class are:

Attr *header*, the column header. If None, default to *_(colid)*

Attr *addcount*, if True, add the table size in parenthesis beside the header

Attr *trheader*, should the header be translated

Attr *escapeheader*, should the header be xml_escaped

Attr *sortable*, tell if the column is sortable

Attr *view*, the table view

Attr *_cw*, the request object

Attr *colid*, the column identifier

Attr *attributes*, dictionary of attributes to put on the HTML tag when the cell is rendered

You may also want to override:

table_size

Return the number of rows (header excluded) to be displayed.

By default return the number of rows in the view's result set. If your table isn't result set based, override this method.

The *has_headers* boolean attribute tells if the table has some headers to be displayed. Default to *True*.

build_column_renderers()

Return a list of column renderers, one for each column to be rendered. Prototype of a column renderer is described below:

```
class AbstractColumnRenderer(header=None, addcount=False, trheader=True,
                               escapeheader=True, sortable=True)
```

Abstract base class for column renderer. Interface of a column renderer follows:

AbstractColumnRenderer.bind(*view*, *colid*)

Bind the column renderer to its view. This is where *_cw*, *view*, *colid* are set and the method to override if you want to add more view/request depending attributes on your column render.

AbstractColumnRenderer.render_header(*w*)

Write label for the specified column by calling *w()*.

AbstractColumnRenderer.render_cell(*w*, *rownum*)

Write value for the specified cell by calling *w()*.

Parameters **rownum** – the row number in the table

AbstractColumnRenderer.sortvalue(*_rownum*)

Return typed value to be used for sorting on the specified column.

Parameters **rownum** – the row number in the table

Attributes on this base class are:

Attr *header*, the column header. If None, default to *_(colid)*

Attr *addcount*, if True, add the table size in parenthesis beside the header

Attr *trheader*, should the header be translated

Attr *escapeheader*, should the header be xml_escaped

Attr *sortable*, tell if the column is sortable

Attr *view*, the table view

Attr *_cw*, the request object

Attr *colid*, the column identifier

Attr *attributes*, dictionary of attributes to put on the HTML tag when the cell is rendered

column_renderer(*colid*, **args*, ***kwargs*)

Return a column renderer for column of the given id.

domid

<wrapped by the cachedproperty decorator>

table_actions()

Return a list of actions (Link) that match the view's result set, and return those in the 'mainactions' category.

property table_size

Return the number of rows (header excluded) to be displayed.

By default return the number of rows in the view's result set. If your table isn't result set based, override this method.

Example

Let us take an example from the timesheet cube:

```
class ActivityResourcesTable(EntityView):
    __regid__ = 'activity.resources.table'
    __select__ = is_instance('Activity')

    def call(self, showresource=True):
        eids = ','.join(str(row[0]) for row in self.cw_rset)
        rql = ('Any R,D,DUR,WO,DESCR,S,A, SN,RT,WT ORDERBY D DESC '
              'WHERE '
              '  A is Activity, A done_by R, R title RT, '
              '  A diem D, A duration DUR, '
              '  A done_for WO, WO title WT, '
              '  A description DESCR, A in_state S, S name SN, '
              '  A eid IN (%s)' % eids)
        rset = self._cw.execute(rql)
        self.wview('resource.table', rset, 'null')

class ResourcesTable(RsetTableView):
    __regid__ = 'resource.table'
    # notice you may wish a stricter selector to check rql's shape
    __select__ = is_instance('Resource')
    # my table headers
    headers = ['Resource', 'diem', 'duration', 'workpackage', 'description', 'state']
    # I want a table where attributes are editable (reedit inside)
    finalvid = 'editable-final'
```

(continues on next page)

(continued from previous page)

```
cellvids = {3: 'editable-final'}
# display facets and actions with a menu
layout_args = {'display_filter': 'top',
               'add_view_actions': None}
```

To obtain an editable table, you may specify the ‘editable-table’ view identifier using some of *cellvids*, *finalvid* or *nonfinalvid*.

The previous example results in:

date	duration	workpackage	description	state	
2010/09/15	1.000	AAA	work on AAA	pending	2010/09/15 leo 1.0 AAA [pending]
2010/09/16	1.000	AAA	continue on AAA	pending	2010/09/16 leo 1.0 AAA [pending]

In order to activate table filter mechanism, the *display_filter* option is given as a layout argument. A small arrow will be displayed at the table’s top right corner. Clicking on *show filter form* action, will display the filter form as below:

has text

duration
1.0 - 1.0

date
2010/09/15 - 2010/09/17

activity (2)	date	done by	duration	done for	description	in	hide filter form
2010/09/15 leo 1.0 AAA [pending]	2010/09/15	Leo	1.000	AAA	work on AAA	per	
2010/09/16 leo 1.0 AAA [pending]	2010/09/16	Leo	1.000	AAA	continue on AAA	pending	

By the same way, you can display additional actions for the selected entities by setting *add_view_actions* layout option to *True*. This will add actions returned by the view’s *table_actions()*.

You can notice that all columns of the result set are not displayed. This is because of given *headers*, implying to display only columns from 0 to *len(headers)*.

Also Notice that the *ResourcesTable* view relies on a particular *rql* shape (which is not ensured by the way, the only checked thing is that the result set contains instance of the *Resource* type). That usually implies that you can’t use this view for user specific queries (e.g. generated by facets or typed manually).

So another option would be to write this view using *EntityTableView*, as below.

```
class ResourcesTable(EntityTableView):
    __regid__ = 'resource.table'
    __select__ = is_instance('Resource')
    # table columns definition
    columns = ['resource', 'diem', 'duration', 'workpackage', 'description', 'in_state']
    # I want a table where attributes are editable (reedit inside)
    finalvid = 'editable-final'
    # display facets and actions with a menu
    layout_args = {'display_filter': 'top',
                  'add_view_actions': None}

    def workpackage_cell(entity):
        activity = entity.reverse_done_in[0]
        activity.view('reedit', rtype='done_for', role='subject', w=w)
```

(continues on next page)

(continued from previous page)

```

def workpackage_sortvalue(entity):
    activity = entity.reverse_done_in[0]
    return activity.done_for[0].sortvalue()

column_renderers = {
    'resource': MainEntityColRenderer(),
    'workpackage': EntityTableColRenderer(
        header='Workpackage',
        renderfunc=workpackage_cell,
        sortfunc=workpackage_sortvalue,),
    'in_state': EntityTableColRenderer(
        renderfunc=lambda w,x: w(x.cw_adapt_to('IWorkflowable').printable_state),
        sortfunc=lambda x: x.cw_adapt_to('IWorkflowable').printable_state),
}

```

Notice the following point:

- *cell_<column>(w, entity)* will be searched for rendering the content of a cell. If not found, *column* is expected to be an attribute of *entity*.
- *cell_sortvalue_<column>(entity)* should return a typed value to use for javascript sorting or None for not sortable columns (the default).
- The *etable_entity_sortvalue()* decorator will set a 'sortvalue' function for the column containing the main entity (the one given as argument to all methods), which will call *entity.sortvalue()*.
- You can set a column header using the *etable_header_title()* decorator. This header will be translated. If it's not an already existing msgid, think to mark it using *_()* (the example supposes headers are schema defined msgid).

Pro/cons of each approach

EntityTableView and *RsetableView* provides basically the same set of features, though they don't share the same properties. Let's try to sum up pro and cons of each class.

- *EntityTableView* view is:
 - more verbose, but usually easier to understand
 - easily extended (easy to add/remove columns for instance)
 - doesn't rely on a particular rset shape. Simply give it a title and will be listed in the 'possible views' box if any.
- *RsetTableView* view is:
 - hard to beat to display barely a result set, or for cases where some of *headers*, *displaycols* or *cellvids* could be defined to enhance the table while you don't care about e.g. pagination or facets.
 - hardly extensible, as you usually have to change places where the view is called to modify the RQL (hence the view's result set shape).

6.5.9 XML and RSS views

(cubicweb.web.views.xmlrss)

Overview

rss Creates a RSS/XML view and call the view *rssitem* for each entity of the result set.

rssitem Create a RSS/XML view for each entity based on the results of the dublin core methods of the entity (*dc_**)

RSS Channel Example

Assuming you have several blog entries, click on the title of the search box in the left column. A larger search box should appear. Enter:

```
Any X ORDERBY D WHERE X is BlogEntry, X creation_date D
```

and you get a list of blog entries.

Click on your login at the top right corner. Chose “user preferences”, then “boxes”, then “possible views box” and check “visible = yes” before validating your changes.

Enter the same query in the search box and you will see the same list, plus a box titled “possible views” in the left column. Click on “entityview”, then “RSS”.

You just applied the “RSS” view to the RQL selection you requested.

That’s it, you have a RSS channel for your blog.

Try again with:

```
Any X ORDERBY D WHERE X is BlogEntry, X creation_date D,
X entry_of B, B title "MyLife"
```

Another RSS channel, but a bit more focused.

A last one for the road:

```
Any C ORDERBY D WHERE C is Comment, C creation_date D LIMIT 15
```

displayed with the RSS view, that’s a channel for the last fifteen comments posted.

[WRITE ME]

- show that the RSS view can be used to display an ordered selection of blog entries, thus providing a RSS channel
- show that a different selection (by category) means a different channel

6.5.10 URL publishing

(*cubicweb.web.views.urlpublishing*)

Associate url's path to view identifier / rql queries.

CubicWeb finds all registered `URLPathEvaluator`s, orders them according to their `priority` attribute and calls their `evaluate_path()` method. The first that returns something and doesn't raise a `PathDontMatch` exception wins.

Here is the default evaluator chain:

1. *cubicweb.web.views.urlpublishing.RawPathEvaluator* handles unique url segments that match exactly one of the registered controller's `__regid__`. Urls such as `/view?`, `/edit?`, `/json?` fall in that category;
2. *cubicweb.web.views.urlpublishing.EidPathEvaluator* handles unique url segments that are eids (e.g. `/1234`);
3. *cubicweb.web.views.urlpublishing.URLRewriteEvaluator* selects all urlrewriter components, sorts them according to their priority, call their `rewrite()` method, the first one that doesn't raise a `KeyError` wins. This is where the *cubicweb.web.views.urlrewrite* and *cubicweb.web.views.urlrewrite.SimpleReqRewriter* comes into play;
4. *cubicweb.web.views.urlpublishing.RestPathEvaluator* handles urls based on entity types and attributes : `<type>((/<attribute name>)]?<attribute value>)?` This is why `cwuser/carlos` works;
5. *cubicweb.web.views.urlpublishing.ActionPathEvaluator* handles any of the previous paths with an additional trailing `"<action>"` segment, `<action>` being one of the registered actions' `__regid__`.

Note: Actionpath executes a query whose results is lost because of redirecting instead of direct traversal.

class *cubicweb.web.views.urlpublishing.URLPublisherComponent* (*vreg*, *default_method*='view')

Associate url path to view identifier / rql queries, by applying a chain of `urlpathevaluator` components.

An evaluator is a `URLPathEvaluator` subclass with an `.evaluate_path` method taking the request object and the path to publish as argument. It will either return a publishing method identifier and an rql query on success or raise a `PathDontMatch` exception on failure. URL evaluators are called according to their `priority` attribute, with 0 as the greatest priority and greater values as lower priority. The first evaluator returning a result or raising something else than `PathDontMatch` will stop the handlers chain.

process(*req*, *path*)

Given a URL (essentially characterized by a path on the server, but additional information may be found in the request object), return a publishing method identifier (e.g. controller) and an optional result set.

Parameters

- **req** (*cubicweb.web.request.CubicWebRequestBase*) – the request object
- **path** (*str*) – the path of the resource to publish. If empty, `None` or `"/` "view" is used as the default path.

Return type tuple(*str*, *cubicweb.rset.ResultSet* or `None`)

Returns the publishing method identifier and an optional result set

Raises *NotFound* – if no handler is able to decode the given path

You can write your own `URLPathEvaluator` class to handle custom paths. For instance, if you want `/my-card-id` to redirect to the corresponding card's primary view, you would write:

```

class CardWikiidEvaluator(URLPathEvaluator):
    priority = 3 # make it be evaluated *before* RestPathEvaluator

    def evaluate_path(self, req, segments):
        if len(segments) != 1:
            raise PathDontMatch()
        rset = req.execute('Any C WHERE C wikiid %(w)s',
                           {'w': segments[0]})
        if len(rset) == 0:
            # Raise NotFound if no card is found
            raise PathDontMatch()
        return None, rset

```

On the other hand, you can also deactivate some of the standard evaluators in your final application. The only thing you have to do is to unregister them, for instance in a *registration_callback* in your cube:

```

def registration_callback(vreg):
    vreg.unregister(RestPathEvaluator)

```

You can even replace the `cubicweb.web.views.urlpublishing.URLPublisherComponent` class if you want to customize the whole toolchain process or if you want to plug into an early enough extension point to control your request parameters:

```

class SanitizerPublisherComponent(URLPublisherComponent):
    """override default publisher component to explicitly ignore
    unauthorized request parameters in anonymous mode.
    """
    unauthorized_form_params = ('rql', 'vid', '__login', '__password')

    def process(self, req, path):
        if req.session.anonymous_session:
            self._remove_unauthorized_params(req)
        return super(SanitizerPublisherComponent, self).process(req, path)

    def _remove_unauthorized_params(self, req):
        for param in req.form.keys():
            if param in self.unauthorized_form_params:
                req.form.pop(param)

def registration_callback(vreg):
    vreg.register_and_replace(SanitizerPublisherComponent, URLPublisherComponent)

```

```

class cubicweb.web.views.urlpublishing.RawPathEvaluator(urlpublisher)
    handle path of the form:

```

```
<publishing_method>?parameters...
```

```

class cubicweb.web.views.urlpublishing.EidPathEvaluator(urlpublisher)
    handle path with the form:

```

```
<eid>
```

```
class cubicweb.web.views.urlpublishing.URLRewriteEvaluator(urlpublisher)
```

tries to find a rewrite rule to apply

URL rewrite rule definitions are stored in URLRewriter objects

```
class cubicweb.web.views.urlpublishing.RestPathEvaluator(urlpublisher)
```

handle path with the form:

```
<etype>[[</attribute name>]/<attribute value>]*
```

```
class cubicweb.web.views.urlpublishing.ActionPathEvaluator(urlpublisher)
```

handle path with the form:

```
<any evaluator path>/<action>
```

6.5.11 URL rewriting

(*cubicweb.web.views.urlrewrite*)

```
class cubicweb.web.views.urlrewrite.URLRewriter(req, **extra)
```

Base class for URL rewriters.

Url rewriters should have a *rules* dict that maps an input URI to something that should be used for rewriting.

The actual logic that defines how the rules dict is used is implemented in the *rewrite* method.

A *priority* attribute might be used to indicate which rewriter should be tried first. The higher the priority is, the earlier the rewriter will be tried.

```
class cubicweb.web.views.urlrewrite.SimpleReqRewriter(req, **extra)
```

The SimpleReqRewriters uses a *rules* dict that maps input URI (regexp or plain string) to a dictionary to update the request's form.

If the input uri is a regexp, group substitution is allowed.

```
rewrite(req, uri)
```

for each *input*, *output* in *rules*, if *uri* matches *input*, req's form is updated with *output*

```
class cubicweb.web.views.urlrewrite.SchemaBasedRewriter(req, **extra)
```

Here, the rules dict maps regexps or plain strings to callbacks that will be called with inputurl, uri, req, schema as parameters.

SimpleReqRewriter is enough for a certain number of simple cases. If it is not sufficient, SchemaBasedRewriter allows to do more elaborate things.

Here is an example of SimpleReqRewriter usage with plain string:

```
from cubicweb.web.views.urlrewrite import SimpleReqRewriter
class TrackerSimpleReqRewriter(SimpleReqRewriter):
    rules = [
        ('/versions', dict(vid='versionsinfo')),
    ]
```

When the url is *<base_url>/versions*, the view with the `__regid__ versionsinfo` is displayed.

Here is an example of SimpleReqRewriter usage with regular expressions:

```

from cubicweb.web.views.urlrewrite import (
    SimpleReqRewriter, rgx)

class BlogReqRewriter(SimpleReqRewriter):
    rules = [
        (rgx('/blogentry/([a-z_]+)\.rss'),
         dict(rql=('Any X ORDERBY CD DESC LIMIT 20 WHERE X is BlogEntry, '
                  'X creation_date CD, X created_by U, '
                  'U login "%(user)s"'
                  % {'user': r'\1'}), vid='rss')))
    ]

```

When a url matches the regular expression, the view with the `__regid__` `rss` which match the result set is displayed.

To deal with URL rewriting with an underlying RQL query, it is possible to specify the behaviour in the case of an empty rset with the option `empty_rset_raises_404`.

The following example shows a `SimpleReqRewriter` usage with the `empty_rset_raises_404` option set to `True`. In this case, the path `mycwetypeurl/pouet` will return a 404. Without this option, it would return a 200.

```

from cubicweb.web.views.urlrewrite import (
    SimpleReqRewriter, rgx)

class MyRewriter(SimpleReqRewriter):
    rules = [(rgx(r'/mycwetypeurl/([^/]+)'),
              dict(vid='primary',
                    rql=r'Any T WHERE T is CWEType, T name "\1"',
                    empty_rset_raises_404=True),)]

```

Here is an example of `SchemaBasedRewriter` usage:

```

from cubicweb.web.views.urlrewrite import (
    SchemaBasedRewriter, rgx, build_rset)

class TrackerURLRewriter(SchemaBasedRewriter):
    rules = [
        (rgx('/project/([^/]+)/([^/]+)/tests'),
         build_rset(rql='Version X WHERE X version_of P, P name %(project)s, X num
↪ %(num)s',
                    rgxgroups=[('project', 1), ('num', 2)], vid='versiontests')),
    ]

```

6.5.12 Breadcrumbs

Breadcrumbs are a navigation component to help the user locate himself along a path of entities.

Display

Breadcrumbs are displayed by default in the header section (see *Layout and sections*). With the default main template, the header section is composed by the logo, the application name, breadcrumbs and, at the most right, the login box. Breadcrumbs are displayed just next to the application name, thus they begin with a separator.

Here is the header section of the CubicWeb's forge:



There are three breadcrumbs components defined in `cubicweb.web.views.ibreadcrumbs`:

- *BreadCrumbEntityVComponent*: displayed for a result set with one line if the entity is adaptable to `IBreadCrumbsAdapter`.
- *BreadCrumbETypeVComponent*: displayed for a result set with more than one line, but with all entities of the same type which can adapt to `IBreadCrumbsAdapter`.
- *BreadCrumbAnyRSetVComponent*: displayed for any other result set.

Building breadcrumbs

The `IBreadCrumbsAdapter` adapter is defined in the `cubicweb.web.views.ibreadcrumbs` module. It specifies that an entity which implements this interface must have a `breadcrumbs` and a `parent_entity` method. A default implementation for each is provided. This implementation exploits the `ITreeAdapter`.

Note: Redefining the breadcrumbs is the hammer way to do it. Another way is to define an *ITreeAdapter* adapter on an entity type. If available, it will be used to compute breadcrumbs.

Here is the API of the `IBreadCrumbsAdapter` class:

`IBreadCrumbsAdapter.parent_entity()`

`IBreadCrumbsAdapter.breadcrumbs(view=None, recurs=None)`

return a list containing some:

- tuple (url, label)
- entity
- simple label string

defining path from a root to the current view

the main view is given as argument so breadcrumbs may vary according to displayed view (may be None). When recursing on a parent entity, the *recurs* argument should be a set of already traversed nodes (infinite loop safety belt).

If the breadcrumbs method return a list of entities, the `cubicweb.web.views.ibreadcrumbs.BreadCrumbView` is used to display the elements.

By default, for any entity, if `recurs=True`, breadcrumbs method returns a list of entities, else a list of a simple string.

In order to see a hierarchical breadcrumbs, entities must have a `parent` method which returns the parent entity. By default this method doesn't exist on entity, given that it can not be guessed.

6.5.13 The 'download' views

Specific views for entities adapting to `IDownloadable`

Components

```
class cubicweb.web.views.idownloadable.DownloadBox(*args, **kwargs)
    add download box
```

Download views

```
class cubicweb.web.views.idownloadable.DownloadView(req=None, rset=None, **kwargs)
    download view

    this view is replacing the deprecated 'download' controller and allow downloading of entities providing the
    necessary interface

class cubicweb.web.views.idownloadable.DownloadLinkView(req=None, rset=None, **kwargs)
    view displaying a link to download the file

class cubicweb.web.views.idownloadable.IDownloadablePrimaryView(req=None, rset=None,
                                                                **kwargs)

class cubicweb.web.views.idownloadable.IDownloadableOneLineView(req=None, rset=None,
                                                                **kwargs)
```

Embedded views

```
class cubicweb.web.views.idownloadable.ImageView(req=None, rset=None, **kwargs)
    image embedded view

class cubicweb.web.views.idownloadable.EHTMLView(req=None, rset=None, **kwargs)
    html embedded view
```

6.5.14 Online documentation system

inline help system, rendering ReST files in the *wdoc* subdirectory of CubicWeb and cubes

Help views

```
class cubicweb.web.views.wdoc.InlineHelpView(req=None, rset=None, **kwargs)
```

Actions

```
class cubicweb.web.views.wdoc.HelpAction(req, **extra)
```

```
class cubicweb.web.views.wdoc.AboutAction(req, **extra)
```

6.6 Configuring the user interface

6.6.1 Relation tags

A RelationTag object is an object which allows to link a configuration information to a relation definition. For instance, the standard primary view uses a RelationTag object (uicfg.primaryview_section) to get the section to display relations.

```
# display `entry_of` relations in the `relations` section in the `BlogEntry` primary_
↪view
uicfg.primaryview_section.tag_subject_of(('BlogEntry', 'entry_of', '*'),
                                         'relations')

# hide every relation `entry_of` in the `Blog` primary view
uicfg.primaryview_section.tag_object_of('*', 'entry_of', 'Blog'), 'hidden')
```

Three primitives are defined:

- tag_subject_of tag a relation in the subject's context
- tag_object_of tag a relation in the object's context
- tag_attribute shortcut for tag_subject_of

6.6.2 The uicfg module

Note: The part of uicfg that deals with primary views is in the *Primary view configuration* chapter.

This module (cubicweb.web.views.uicfg) regroupes a set of structures that may be used to configure various options of the generated web interface.

To configure the interface generation, we use RelationTag objects.

Index view configuration

indexview_etype_section entity type category in the index/manage page. May be one of:

- application
- system
- schema
- subobject (not displayed by default)

By default only entities on the application category are shown.


```
from cubicweb.web.views import uicfg
# force hiding
uicfg.indexview_etype_section['HideMe'] = 'subobject'
# force display
uicfg.indexview_etype_section['ShowMe'] = 'application'
```

Actions box configuration

actionbox_appearsin_addmenu simple boolean relation tags used to control the “add entity” submenu. Relations whose rtag is True will appear, other won’t.

```
# Adds all subjects of the entry_of relation in the add menu of the ``Blog``
# primary view
uicfg.actionbox_appearsin_addmenu.tag_object_of('*', 'entry_of', 'Blog'), True)
```

6.6.3 The uihelper module

This module provide highlevel helpers to avoid uicfg boilerplate for most common tasks such as fields ordering, widget customization, etc.

Here are a few helpers to customize *action box* rendering:

`cubicweb.web.uihelper.append_to_addmenu(etype, attr, createdtype=*)` → Callable

`cubicweb.web.uihelper.remove_from_addmenu(etype, attr, createdtype=*)` → Callable

and a few other ones for *form configuration*:

`cubicweb.web.uihelper.set_fields_order(etype, attrs)` → Callable

`cubicweb.web.uihelper.hide_field(etype, attr, desttype=*, formtype='main')` → Callable

`cubicweb.web.uihelper.hide_fields(etype, attrs, formtype='main')` → Callable

`cubicweb.web.uihelper.set_field_kwargs(etype, attr, **kwargs)` → Callable

`cubicweb.web.uihelper.set_field(etype, attr, field)` → Callable

`cubicweb.web.uihelper.edit_inline(etype, attr, desttype=*, formtype=('main', 'inlined'))` → Callable

`cubicweb.web.uihelper.edit_as_attr(etype, attr, desttype=*, formtype=('main', 'muledit'))` → Callable

`cubicweb.web.uihelper.set_muledit_editable(etype, attrs)` → Callable

The module also provides a *FormConfig* base class that lets you gather uicfg declaration in the scope of a single class, which can sometimes be clearer to read than a bunch of sequential function calls.

class `cubicweb.web.uihelper.FormConfig`

helper base class to define uicfg rules on a given entity type.

In all descriptions below, attributes list can either be a list of attribute names or a list of 2-tuples (relation name, role of the edited entity in the relation).

Attributes

etype which entity type the form config is for. This attribute is **mandatory**

formtype the formtype the class tries to customize (i.e. *main*, *inlined*, or *muledit*), default is *main*.

hidden the list of attributes or relations to hide.

rels_as_attrs the list of attributes to edit in the *attributes* section.

inlined the list of attributes to edit in the *inlined* section.

fields_order the list of attributes to edit, in the desired order. Unspecified fields will be displayed after specified ones, their order being consistent with the schema definition.

widgets a dictionary mapping attribute names to widget instances.

fields a dictionary mapping attribute names to field instances.

uicfg_afs an instance of `cubicweb.web.uicfg.AutoformSectionRelationTags` Default is `None`, meaning `cubicweb.web.uicfg.autoform_section` is used.

uicfg_aff an instance of `cubicweb.web.uicfg.AutoformFieldTags` Default is `None`, meaning `cubicweb.web.uicfg.autoform_field` is used.

uicfg_affk an instance of `cubicweb.web.uicfg.AutoformFieldKwargsTags` Default is `None`, meaning `cubicweb.web.uicfg.autoform_field_kwargs` is used.

Examples:

```
from cubicweb.web import uihelper, formwidgets as fwdgs

class LinkFormConfig(uihelper.FormConfig):
    etype = 'Link'
    hidden = ('title', 'description', 'embed')
    widgets = dict(
        url=fwdgs.TextInput(attrs={'size':40}),
    )

class UserFormConfig(uihelper.FormConfig):
    etype = 'CWUser'
    hidden = ('login',)
    rels_as_attrs = ('in_group',)
    fields_order = ('firstname', 'surname', 'in_group', 'use_email')
    inlined = ('use_email',)
```

6.7 Ajax

Warning: This approach is deprecated in favor of using `cwclientlibjs`. If you use react for your UI, try the react components from the `cwelements` library. The documentation is kept here as reference.

For historical reference of what Ajax is and used to be, one can read the [wikipedia article about Ajax](#).

CubicWeb provides a few helpers to facilitate *javascript* <-> *python* communications.

You can, for instance, register some python functions that will become callable from javascript through ajax calls. All the ajax URLs are handled by the `cubicweb.web.views.ajaxcontroller.AjaxController` controller.

The `ajaxcontroller` module defines the `AjaxController` controller and the `ajax-func` cubicweb registry.

class cubicweb.web.views.ajaxcontroller.**AjaxController**(*args, **kwargs)

AjaxController handles ajax remote calls from javascript

The following javascript function call:

```
var d = asyncRemoteExec('foo', 12, "hello");
d.addCallback(function(result) {
    alert('server response is: ' + result);
});
```

will generate an ajax HTTP GET on the following url:

```
BASE_URL/ajax?fname=foo&arg=12&arg="hello"
```

The AjaxController controller will therefore be selected to handle those URLs and will itself select the `cubicweb.web.views.ajaxcontroller.AjaxFunction` matching the `fname` parameter.

publish(rset=None)

publish the current request, with an optional input rset

ajax-funcs registry hosts exposed remote functions, that is functions that can be called from the javascript world.

To register a new remote function, either decorate your function with the `ajaxfunc()` decorator:

```
from cubicweb.predicates import match_user_groups
from cubicweb.web.views.ajaxcontroller import ajaxfunc

@ajaxfunc(output_type='json', selector=match_user_groups('managers'))
def list_users(self):
    return [u for (u,) in self._cw.execute('Any L WHERE U login L')]
```

or inherit from `AjaxFunction` and implement the `__call__` method:

```
from cubicweb.web.views.ajaxcontroller import AjaxFunction
class ListUser(AjaxFunction):
    __regid__ = 'list_users' # __regid__ is the name of the exposed function
    __select__ = match_user_groups('managers')
    output_type = 'json'

    def __call__(self):
        return [u for (u, ) in self._cw.execute('Any L WHERE U login L')]
```

class cubicweb.web.views.ajaxcontroller.**AjaxFunction**(req, **extra)

Attributes on this base class are:

Attr `check_pageid`: make sure the pageid received is valid before proceeding

Attr `output_type`:

- *None*: no processing, no change on content-type
- *json*: serialize with `json.dumps` and set `application/json` content-type
- *xhtml*: wrap result in an XML node and forces `HTML / XHTML` content-type (use `_cw.html_content_type()`)

cubicweb.web.views.ajaxcontroller.**ajaxfunc**(implementation=None, selector=<Predicate yes at 7fe6a7b56590>, output_type=None, check_pageid=False, regid=None)

promote a standard function to an `AjaxFunction` appobject.

All parameters are optional:

Parameters

- **selector** – a custom selector object if needed, default is `yes()`
- **output_type** – either `None`, `'json'` or `'xhtml'` to customize output content-type. Default is `None`
- **check_pageid** – whether the function requires a valid *pageid* or not to proceed. Default is `False`.
- **regid** – a custom `__regid__` for the created `AjaxFunction` object. Default is to keep the wrapped function name.

`ajaxfunc` can be used both as a standalone decorator:

```
@ajaxfunc
def my_function(self):
    return 42
```

or as a parametrizable decorator:

```
@ajaxfunc(output_type='json')
def my_function(self):
    return 42
```

6.8 Javascript

CubicWeb uses quite a bit of javascript in its user interface and ships with jquery (1.3.x) and parts of the jquery UI library, plus a number of homegrown files and also other third party libraries.

All javascript files are stored in `cubicweb/web/data/`. There are around thirty js files there. In a cube it goes to `data/`.

Obviously one does not want javascript pieces to be loaded all at once, hence the framework provides a number of mechanisms and conventions to deal with javascript resources.

6.8.1 Conventions

It is good practice to name cube specific js files after the name of the cube, like this : `'cube.mycube.js'`, so as to avoid name clashes.

6.8.2 Server-side Javascript API

Javascript resources are typically loaded on demand, from views. The request object (available as `self._cw` from most application objects, for instance views and entities objects) has a few methods to do that:

- `add_js(self, jsfiles, localfile=True)` which takes a sequence of javascript files and writes proper entries into the HTML header section. The `localfile` parameter allows to declare resources which are not from `web/data` (for instance, residing on a content delivery network).
- `add_onload(self, jscode)` which adds one raw javascript code snippet inline in the html headers. This is quite useful for setting up early `jQuery(document).ready(...)` initialisations.

6.8.3 Javascript events

- **server-response**: this event is triggered on HTTP responses (both standard and ajax). The two following extra parameters are passed to callbacks :
 - **ajax**: a boolean that says if the reponse was issued by an ajax request
 - **node**: the DOM node returned by the server in case of an ajax request, otherwise the document itself for standard HTTP requests.

6.8.4 Important javascript AJAX APIS

- *asyncRemoteExec* and *remoteExec* are the base building blocks for doing arbitrary async (resp. sync) communications with the server
- *reloadComponent* is a convenience function to replace a DOM node with server supplied content coming from a specific registry (this is quite handy to refresh the content of some boxes for instances)
- *jQuery.fn.loadxhtml* is an important extension to jQuery which allows proper loading and in-place DOM update of xhtml views. It is suitably augmented to trigger necessary events, and process CubicWeb specific elements such as the facet system, fckeditor, etc.

6.8.5 A simple example with asyncRemoteExec

On the python side, we have to define an `cubicweb.web.views.ajaxcontroller.AjaxFunction` object. The simplest way to do that is to use the `cubicweb.web.views.ajaxcontroller.ajaxfunc()` decorator (for more details on this, refer to *Ajax*).

On the javascript side, we do the asynchronous call. Notice how it creates a *deferred* object. Proper treatment of the return value or error handling has to be done through the `addCallback` and `addErrback` methods.

6.8.6 Anatomy of a reloadComponent call

reloadComponent allows to dynamically replace some DOM node with new elements. It has the following signature:

- *compid* (mandatory) is the name of the component to be reloaded
- *rql* (optional) will be used to generate a result set given as argument to the selected component
- *registry* (optional) defaults to 'components' but can be any other valid registry name
- *nodeid* (optional) defaults to compid + 'Component' but can be any explicitly specified DOM node id
- *extraargs* (optional) should be a dictionary of values that will be given to the `cell_call` method of the component

6.8.7 A simple reloadComponent example

The server side implementation of *reloadComponent* is the `cubicweb.web.views.ajaxcontroller.component()` *AjaxFunction* appobject.

The following function implements a two-steps method to delete a standard bookmark and refresh the UI, while keeping the UI responsive.

```
function removeBookmark(beid) {
    d = asyncRemoteExec('delete_bookmark', beid);
    d.addCallback(function(boxcontent) {
        reloadComponent('bookmarks_box', '', 'boxes', 'bookmarks_box');
        document.location.hash = '#header';
        updateMessage(_("bookmark has been removed"));
    });
}
```

reloadComponent is called with the id of the bookmark box as argument, no *rql* expression (because the bookmarks display is actually independant of any dataset context), a reference to the ‘boxes’ registry (which hosts all left, right and contextual boxes) and finally an explicit ‘bookmarks_box’ nodeid argument that stipulates the target DOM node.

6.8.8 Anatomy of a loadxhtml call

jQuery.fn.loadxhtml is an important extension to jQuery which allows proper loading and in-place DOM update of xhtml views. The existing *jQuery.load* function does not handle xhtml, hence the addition. The API of *loadxhtml* is roughly similar to that of *jQuery.load*.

- *url* (mandatory) should be a complete url (typically referencing the *cubicweb.web.views.ajaxcontroller.AjaxController*, but this is not strictly mandatory)
- *data* (optional) is a dictionary of values given to the controller specified through an *url* argument; some keys may have a special meaning depending on the choosen controller (such as *fname* for the *JsonController*); the *callback* key, if present, must refer to a function to be called at the end of *loadxhtml* (more on this below)
- *reqtype* (optional) specifies the request method to be used (get or post); if the argument is ‘post’, then the post method is used, otherwise the get method is used
- *mode* (optional) is one of *replace* (the default) which means the loaded node will replace the current node content, *swap* to replace the current node with the loaded node, and *append* which will append the loaded node to the current node content

About the *callback* option:

- it is called with two parameters: the current node, and a list containing the loaded (and post-processed node)
- whenever it returns another function, this function is called in turn with the same parameters as above

This mechanism allows callback chaining.

6.8.9 A simple example with loadxhtml

Here we are concerned with the retrieval of a specific view to be injected in the live DOM. The view will be of course selected server-side using an entity *eid* provided by the client side.

```
from cubicweb.web.views.ajaxcontroller import ajaxfunc

@ajaxfunc(output_type='xhtml')
def frob_status(self, eid, frobname):
    entity = self._cw.entity_from_eid(eid)
    return entity.view('frob', name=frobname)
```

```
function updateSomeDiv(divid, eid, frobname) {
    var params = {fname:'frob_status', eid: eid, frobname:frobname};
    jQuery('#'+divid).loadxhtml(JSON_BASE_URL, params, 'post');
}
```

In this example, the url argument is the base json url of a cube instance (it should contain something like *http://myinstance/ajax?*). The actual AjaxController method name is encoded in the *params* dictionary using the *fname* key.

6.8.10 A more real-life example

A frequent need of Web 2 applications is the delayed (or demand driven) loading of pieces of the DOM. This is typically achieved using some preparation of the initial DOM nodes, jQuery event handling and proper use of *loadxhtml*.

We present here a skeletal version of the mechanism used in CubicWeb and available in *web/views/tabs.py*, in the *LazyViewMixin* class.

```
def lazyview(self, vid, rql=None):
    """ a lazy version of wview """
    self._cw.add_js('cubicweb.lazy.js')
    urlparams = {'vid' : vid, 'fname' : 'view'}
    if rql is not None:
        urlparams['rql'] = rql
    self.w(u'<div id="lazy-%s" cubicweb:loadurl="%s">',
          vid, xml_escape(self._cw.build_url('json', **urlparams)))
    self.w(u'</div>')
    self._cw.add_onload(u'jQuery("#lazy-%(vid)s").bind("%(event)s", function() {
        loadNow("#lazy-%(vid)s");});');"""
    % {'event': 'load_%s' % vid, 'vid': vid})
```

This creates a *div* with a specific event associated to it.

The full version deals with:

- optional parameters such as an entity *eid*, an *rset*
- the ability to further reload the fragment
- the ability to display a spinning wheel while the fragment is still not loaded
- handling of browsers that do not support ajax (search engines, text-based browsers such as lynx, etc.)

The javascript side is quite simple, due to *loadxhtml* awesomeness.

```
function loadNow(eltset) {
    var lazydiv = jQuery(eltset);
    lazydiv.loadxhtml(lazydiv.attr('cubicweb:loadurl'));
}
```

This is all significantly different of the previous *simple example* (albeit this example actually comes from real-life code).

Notice how the *cubicweb:loadurl* is used to convey the url information. The base of this url is similar to the global javascript *JSON_BASE_URL*. According to the pattern described earlier, the *fname* parameter refers to the standard *js_view* method of the *JSONController*. This method renders an arbitrary view provided a view id (or *vid*) is provided, and most likely an *rql* expression yielding a result set against which a proper view instance will be selected.

The `cubicweb:loadurl` is one of the 29 attributes extensions to XHTML in a specific cubicweb namespace. It is a means to pass information without breaking HTML nor XHTML compliance and without resorting to ugly hacks.

Given all this, it is easy to add a small nevertheless useful feature to force the loading of a lazy view (for instance, a very computation-intensive web page could be scindé into one fast-loading part and a delayed part).

On the server side, a simple call to a javascript function is sufficient.

```
def forceview(self, vid):
    """trigger an event that will force immediate loading of the view
    on dom readiness
    """
    self._cw.add_onload("triggerLoad('%s');" % vid)
```

The browser-side definition follows.

```
function triggerLoad(divid) {
    jQuery('#lazy-' + divid).trigger('load_' + divid);
}
```

6.8.11 Javascript library: overview

- `jquery.*` : jquery and jquery UI library
- `cubicweb.ajax.js` : concentrates all ajax related facilities (it extends jQuery with the `loadhtml` function, provides a handful of high-level ajaxy operations like `asyncRemoteExec`, `reloadComponent`, `replacePageChunk`, `getDomFromResponse`)
- `cubicweb.python.js` : adds a number of practical extension to standard javascript objects (on Date, Array, String, some list and dictionary operations), and a pythonesque way to build classes. Defines a CubicWeb namespace.
- `cubicweb.htmlhelpers.js` : a small bag of convenience functions used in various other cubicweb javascript resources (baseuri, progress cursor handling, popup login box, `html2dom` function, etc.)
- `cubicweb.widgets.js` : provides a widget namespace and constructors and helpers for various widgets (mainly facets and timeline)
- `cubicweb.edition.js` : used by edition forms
- `cubicweb.preferences.js` : used by the preference form
- `cubicweb.facets.js` : used by the facets mechanism

There is also javascript support for massmailing, gmap (google maps), `fcwconfig` (fck editor), timeline, calendar, goa (CubicWeb over AppEngine), flot (charts drawing), tabs and bookmarks.

6.8.12 API

6.8.13 Testing javascript

You with the `cubicweb.qunit.QUnitTestCase` can include standard Qunit tests inside the python unittest run. You simply have to define a new class that inherit from `QUnitTestCase` and register your javascript test file in the `all_js_tests` class attribute. This `all_js_tests` is a sequence a 3-tuple (`<test_file>`, [`<dependencies>`], [`<data_files>`]):

The `<test_file>` should contains the qunit test. `<dependencies>` defines the list of javascript file that must be imported before the test script. Dependencies are included their definition order. `<data_files>` are additional files copied in the test directory. both `<dependencies>` and `<data_files>` are optional. `jquery.js` is preincluded in for all test.


```

from cubicweb.qunit import QUnitTestCase

class MyQUnitTest(QUnitTestCase):

    all_js_tests = (
        ("relative/path/to/my_simple_testcase.js",)
        ("relative/path/to/my_qunit_testcase.js", (
            "rel/path/to/dependency_1.js",
            "rel/path/to/dependency_2.js",)),
        ("relative/path/to/my_complexe_qunit_testcase.js", (
            "rel/path/to/dependency_1.js",
            "rel/path/to/dependency_2.js",
        ), (
            "rel/path/file_dependency.html",
            "path/file_dependency.json")
        ),
    )

```

6.9 CSS Stylesheet

6.9.1 Conventions

6.9.2 Extending / overriding existing styles

We cannot modify the order in which the application is reading the CSS. In the case we want to create new CSS style, the best is to define it in a new CSS located under `myapp/data/` and use those new styles while writing customized views and templates.

If you want to modify an existing CSS styling property, you will have to use `!important` declaration to override the existing property. The application apply a higher priority on the default CSS and you can not change that. Customized CSS will not be read first.

6.9.3 CubicWeb stylesheets

6.10 Edition control

This chapter covers the editing capabilities of *CubicWeb*. It explains html Form construction, the Edit Controller and their interactions.

6.10.1 HTML form construction

CubicWeb provides the somewhat usual form / field / widget / renderer abstraction to provide generic building blocks which will greatly help you in building forms properly integrated with CubicWeb (coherent display, error handling, etc...), while keeping things as flexible as possible.

A `form` basically only holds a set of `fields`, and has to be bound to a `renderer` which is responsible to layout them. Each field is bound to a `widget` that will be used to fill in value(s) for that field (at form generation time) and ‘decode’ (fetch and give a proper Python type to) values sent back by the browser.

The field should be used according to the type of what you want to edit. E.g. if you want to edit some date, you'll have to use the `cubicweb.web.formfields.DateField`. Then you can choose among multiple widgets to edit it, for instance `cubicweb.web.formwidgets.TextInput` (a bare text field), `DateTimePicker` (a simple calendar) or even `jQueryDatePicker` (the JQuery calendar). You can of course also write your own widget.

Exploring the available forms

A small excursion into a *CubicWeb* shell is the quickest way to discover available forms (or application objects in general).

```
>>> from pprint import pprint
>>> pprint( session.vreg['forms'] )
{'base': [<class 'cubicweb.web.views.forms.FieldsForm'>,
          <class 'cubicweb.web.views.forms.EntityFieldsForm'>],
 'changestate': [<class 'cubicweb.web.views.workflow.ChangeStateForm'>,
                  <class 'cubicweb_tracker.views.forms.VersionChangeStateForm'>],
 'composite': [<class 'cubicweb.web.views.forms.CompositeForm'>,
                <class 'cubicweb.web.views.forms.CompositeEntityForm'>],
 'deleteconf': [<class 'cubicweb.web.views.editforms.DeleteConfForm'>],
 'edition': [<class 'cubicweb.web.views.autoformAutomaticEntityForm'>,
              <class 'cubicweb.web.views.workflow.TransitionEditionForm'>,
              <class 'cubicweb.web.views.workflow.StateEditionForm'>],
 'logform': [<class 'cubicweb.web.views.basetemplates.LogForm'>],
 'massmailing': [<class 'cubicweb.web.views.massmailing.MassMailingForm'>],
 'muledit': [<class 'cubicweb.web.views.editforms.TableEditForm'>]}
```

The two most important form families here (for all practical purposes) are *base* and *edition*. Most of the time one wants alterations of the `AutomaticEntityForm` to generate custom forms to handle edition of an entity.

The Automatic Entity Form

class `cubicweb.web.views.autoformAutomaticEntityForm(*args, **kwargs)`

`AutomaticEntityForm` is an automagic form to edit any entity. It is designed to be fully generated from schema but highly configurable through `uicfg`.

Of course, as for other forms, you can also customise it by specifying various standard form parameters on selection, overriding, or adding/removing fields in selected instances.

Configuration through uicfg

It is possible to manage which and how an entity's attributes and relations will be edited in the various contexts where the automatic entity form is used by using proper `uicfg` tags.

The details of the `uicfg` syntax can be found in the *The uicfg module* chapter.

Possible relation tags that apply to entity forms are detailed below. They are all in the `cubicweb.web.uicfg` module.

Attributes/relations display location

`autoform_section` specifies where to display a relation in form for a given form type. `tag_attribute()`, `tag_subject_of()` and `tag_object_of()` methods for this relation tag expect two arguments additionally to the relation key: a *formtype* and a *section*.

formtype may be one of:

- ‘main’, the main entity form (e.g. the one you get when creating or editing an entity)
- ‘inlined’, the form for an entity inlined into another form
- ‘muledit’, the table form when editing multiple entities of the same type

section may be one of:

- ‘hidden’, don’t display (not even in a hidden input)
- ‘attributes’, display in the attributes section
- ‘relations’, display in the relations section, using the generic relation selector combobox (available in main form only, and not usable for attributes)
- ‘inlined’, display target entity of the relation into an inlined form (available in main form only, and not for attributes)

By default, mandatory relations are displayed in the ‘attributes’ section, others in ‘relations’ section.

Change default fields

Use `autoform_field` to replace the default field class to use for a relation or attribute. You can put either a field class or instance as value (put a class whenever it’s possible).

Warning: `autoform_field_kwargs` should usually be used instead of `autoform_field`. If you put a field instance into `autoform_field`, `autoform_field_kwargs` values for this relation will be ignored.

Customize field options

In order to customize field options (see *Field* for a detailed list of options), use `autoform_field_kwargs`. This rtag takes a dictionary as arguments, that will be given to the field’s constructor.

You can then put in that dictionary any arguments supported by the field class. For instance:

```
# Change the content of the combobox. Here `ticket_done_in_choices` is a
# function which returns a list of elements to populate the combobox
autoform_field_kwargs.tag_subject_of(('Ticket', 'done_in', '*'),
                                     {'sort': False,
                                      'choices': ticket_done_in_choices})

# Force usage of a TextInput widget for the expression attribute of
# RQLEExpression entities
autoform_field_kwargs.tag_attribute(('RQLEExpression', 'expression'),
                                     {'widget': fw.TextInput})
```

Note: the widget argument can be either a class or an instance (the later case being convenient to pass the Widget specific initialisation options)

Overriding permissions

The `autoform_permissions_overrides` rtag provides a way to by-pass security checking for dark-corner case where it can't be verified properly.

Anatomy of a choices function

Let's have a look at the `ticket_done_in_choices` function given to the `choices` parameter of the relation tag that is applied to the ('Ticket', 'done_in', '*') relation definition, as it is both typical and sophisticated enough. This is a code snippet from the `tracker` cube.

The Ticket entity type can be related to a Project and a Version, respectively through the `concerns` and `done_in` relations. When a user is about to edit a ticket, we want to fill the combo box for the `done_in` relation with values pertinent with respect to the context. The important context here is:

- creation or modification (we cannot fetch values the same way in either case)
- `__linkto` url parameter given in a creation context

```
from cubicweb.web import formfields

def ticket_done_in_choices(form, field):
    entity = form.edited_entity
    # first see if its specified by __linkto form parameters
    linkedto = form.linked_to[('done_in', 'subject')]
    if linkedto:
        return linkedto
    # it isn't, get initial values
    vocab = field.relvoc_init(form)
    veid = None
    # try to fetch the (already or pending) related version and project
    if not entity.has_eid():
        peids = form.linked_to[('concerns', 'subject')]
        peid = peids and peids[0]
    else:
        peid = entity.project.eid
        veid = entity.done_in and entity.done_in[0].eid
    if peid:
        # we can complete the vocabulary with relevant values
        rschema = form._cw.vreg.schema['done_in'].rdef('Ticket', 'Version')
        rset = form._cw.execute(
            'Any V, VN ORDERBY version_sort_value(VN) '
            'WHERE V version_of P, P eid %(p)s, V num VN, '
            'V in_state ST, NOT ST name "published", {'p': peid}, 'p')
        vocab += [(v.view('combobox'), v.eid) for v in rset.entities()
                  if rschema.has_perm(form._cw, 'add', toeid=v.eid)
                  and v.eid != veid]
    return vocab
```

The first thing we have to do is fetch potential values from the `__linkto` url parameter that is often found in entity creation contexts (the creation action provides such a parameter with a predetermined value; for instance in this case, ticket creation could occur in the context of a *Version* entity). The *RelationField* field class provides a `relvoc_linkedto()` method that gets a list suitably filled with vocabulary values.

```
linkedto = field.relvoc_linkedto(form)
if linkedto:
    return linkedto
```

Then, if no `__linkto` argument was given, we must prepare the vocabulary with an initial empty value (because *done_in* is not mandatory, we must allow the user to not select a version) and already linked values. This is done with the `relvoc_init()` method.

```
vocab = field.relvoc_init(form)
```

But then, we have to give more: if the ticket is related to a project, we should provide all the non published versions of this project (*Version* and *Project* can be related through the *version_of* relation). Conversely, if we do not know yet the project, it would not make sense to propose all existing versions as it could potentially lead to incoherences. Even if these will be caught by some *RQLConstraint*, it is wise not to tempt the user with error-inducing candidate values.

The “ticket is related to a project” part must be decomposed as:

- this is a new ticket which is created in the context of a project
- this is an already existing ticket, linked to a project (through the *concerns* relation)
- there is no related project (quite unlikely given the cardinality of the *concerns* relation, so it can only mean that we are creating a new ticket, and a project is about to be selected but there is no `__linkto` argument)

Note: the last situation could happen in several ways, but of course in a polished application, the paths to ticket creation should be controlled so as to avoid a suboptimal end-user experience

Hence, we try to fetch the related project.

```
veid = None
if not entity.has_eid():
    peids = form.linked_to(['concerns', 'subject'])
    peid = peids and peids[0]
else:
    peid = entity.project.eid
    veid = entity.done_in and entity.done_in[0].eid
```

We distinguish between entity creation and entity modification using the `Entity.has_eid()` method, which returns *False* on creation. At creation time the only way to get a project is through the `__linkto` parameter. Notice that we fetch the version in which the ticket is *done_in* if any, for later.

Note: the implementation above assumes that if there is a `__linkto` parameter, it is only about a project. While it makes sense most of the time, it is not an absolute. Depending on how an entity creation action url is built, several outcomes could be possible there

If the ticket is already linked to a project, fetching it is trivial. Then we add the relevant version to the initial vocabulary.

```
if peid:
    rschema = form._cw.vreg.schema['done_in'].rdef('Ticket', 'Version')
```

(continues on next page)

(continued from previous page)

```

rset = form._cw.execute(
    'Any V, VN ORDERBY version_sort_value(VN) '
    'WHERE V version_of P, P eid %(p)s, V num VN, '
    'V in_state ST, NOT ST name "published"', {'p': peid})
vocab += [(v.view('combobox'), v.eid) for v in rset.entities()
           if rschema.has_perm(form._cw, 'add', toeid=v.eid)
           and v.eid != veid]

```

Warning: we have to defend ourselves against lack of a project eid. Given the cardinality of the *concerns* relation, there *must* be a project, but this rule can only be enforced at validation time, which will happen of course only after form submission

Here, given a project eid, we complete the vocabulary with all unpublished versions defined in the project (sorted by number) for which the current user is allowed to establish the relation.

Building self-posted form with custom fields/widgets

Sometimes you want a form that is not related to entity edition. For those, you'll have to handle form posting by yourself. Here is a complete example on how to achieve this (and more).

Imagine you want a form that selects a month period. There are no proper field/widget to handle this in CubicWeb, so let's start by defining them:

```

# let's have the whole import list at the beginning, even those necessary for
# subsequent snippets
from logilab.common import date
from logilab.mtconverter import xml_escape
from cubicweb.view import View
from cubicweb.predicates import match_kwargs
from cubicweb.web import RequestError, ProcessFormError
from cubicweb.web import formfields as fields, formwidgets as wdgs
from cubicweb.web.views import forms, calendar

class MonthSelect(wdgs.Select):
    """Custom widget to display month and year. Expect value to be given as a
    date instance.
    """

    def format_value(self, form, field, value):
        return u'%s/%s' % (value.year, value.month)

    def process_field_data(self, form, field):
        val = super(MonthSelect, self).process_field_data(form, field)
        try:
            year, month = val.split('/')
            year = int(year)
            month = int(month)
            return date.date(year, month, 1)
        except ValueError:
            raise ProcessFormError(

```

(continues on next page)

(continued from previous page)

```

        form._cw._('badly formatted date string %s') % val)

class MonthPeriodField(fields.CompoundField):
    """custom field composed of two subfields, 'begin_month' and 'end_month'.

    It expects to be used on form that has 'mindate' and 'maxdate' in its
    extra arguments, telling the range of month to display.
    """

    def __init__(self, *args, **kwargs):
        kwargs.setdefault('widget', wdgs.IntervalWidget())
        super(MonthPeriodField, self).__init__(
            [fields.StringField(name='begin_month',
                               choices=self.get_range, sort=False,
                               value=self.get_mindate,
                               widget=MonthSelect()),
             fields.StringField(name='end_month',
                               choices=self.get_range, sort=False,
                               value=self.get_maxdate,
                               widget=MonthSelect())], *args, **kwargs)

    @staticmethod
    def get_range(form, field):
        mindate = date.today(form.cw_extra_kwargs['mindate'])
        maxdate = date.today(form.cw_extra_kwargs['maxdate'])
        assert mindate <= maxdate
        _ = form._cw._
        months = []
        while mindate <= maxdate:
            label = '%s %s' % (_(calendar.MONTHNAMES[mindate.month - 1]),
                               mindate.year)
            value = field.widget.format_value(form, field, mindate)
            months.append( (label, value) )
            mindate = date.next_month(mindate)
        return months

    @staticmethod
    def get_mindate(form, field):
        return form.cw_extra_kwargs['mindate']

    @staticmethod
    def get_maxdate(form, field):
        return form.cw_extra_kwargs['maxdate']

    def process_posted(self, form):
        for field, value in super(MonthPeriodField, self).process_posted(form):
            if field.name == 'end_month':
                value = date.last_day(value)
            yield field, value

```

Here we first define a widget that will be used to select the beginning and the end of the period, displaying months like ‘<month> YYYY’ but using ‘YYYY/mm’ as actual value.

We then define a field that will actually hold two fields, one for the beginning and another for the end of the period. Each subfield uses the widget we defined earlier, and the outer field itself uses the standard `IntervalWidget`. The field adds some logic:

- a vocabulary generation function `get_range`, used to populate each sub-field
- two ‘value’ functions `get_mindate` and `get_maxdate`, used to tell to subfields which value they should consider on form initialization
- overriding of `process_posted`, called when the form is being posted, so that the end of the period is properly set to the last day of the month.

Now, we can define a very simple form:

```
class MonthPeriodSelectorForm(forms.FieldsForm):
    __regid__ = 'myform'
    __select__ = match_kwargs('mindate', 'maxdate')

    form_buttons = [wdgs.SubmitButton()]
    form_renderer_id = 'onerowtable'
    period = MonthPeriodField()
```

where we simply add our field, set a submit button and use a very simple renderer (try others!). Also we specify a selector that ensures form will have arguments necessary to our field.

Now, we need a view that will wrap the form and handle post when it occurs, simply displaying posted values in the page:

```
class SelfPostingForm(View):
    __regid__ = 'myformview'

    def call(self):
        mindate, maxdate = date.date(2010, 1, 1), date.date(2012, 1, 1)
        form = self._cw.vreg['forms'].select(
            'myform', self._cw, mindate=mindate, maxdate=maxdate, action='')
        try:
            posted = form.process_posted()
            self.w(u'<p>posted values %s</p>' % xml_escape(repr(posted)))
        except RequestError: # no specified period asked
            pass
        form.render(w=self.w, formvalues=self._cw.form)
```

Notice usage of the `process_posted()` method, that will return a dictionary of typed values (because they have been processed by the field). In our case, when the form is posted you should see a dictionary with ‘begin_month’ and ‘end_month’ as keys with the selected dates as value (as a python *date* object).

APIs

The Field class and basic fields

Note: Fields are used to control what’s edited in forms. They makes the link between something to edit and its display in the form. Actual display is handled by a widget associated to the field.

Let first see the base class for fields:

class cubicweb.web.formfields.**Field**(*name=None, label=<nullobject>, widget=None, **kwargs*)

This class is the abstract base class for all fields. It hold a bunch of attributes which may be used for fine control of the behaviour of a concrete field.

Attributes

All the attributes described below have sensible default value which may be overridden by named arguments given to field's constructor.

name base name of the field (basestring). The actual input name is returned by the `input_name()` method and may differ from that name (for instance if `eidparam` is true).

id DOM identifier (default to the same value as `name`), should be unique in a form.

label label of the field (default to the same value as `name`).

help help message about this field.

widget widget associated to the field. Each field class has a default widget class which may be overridden per instance.

value field value. May be an actual value or a callable which should take the form and the field as argument and return a value.

choices static vocabulary for this field. May be a list of values, a list of (label, value) tuples or a callable which should take the form and field as arguments and return a list of values or a list of (label, value).

required bool flag telling if the field is required or not.

sort bool flag telling if the vocabulary (either static vocabulary specified in `choices` or dynamic vocabulary fetched from the form) should be sorted on label.

internationalizable bool flag telling if the vocabulary labels should be translated using the current request language.

eidparam bool flag telling if this field is linked to a specific entity

role when the field is linked to an entity attribute or relation, tells the role of the entity in the relation (eg 'subject' or 'object'). If this is not an attribute or relation of the edited entity, `role` should be `None`.

fieldset optional fieldset to which this field belongs to

order key used by automatic forms to sort fields

ignore_req_params when true, this field won't consider value potentially specified using request's form parameters (eg you won't be able to specify a value using for instance url like `http://mywebsite.com/form?field=value`)

Generic methods

input_name(*form, suffix=None*)

Return the 'qualified name' for this field, e.g. something suitable to use as HTML input name. You can specify a suffix that will be included in the name when widget needs several inputs.

dom_id(*form, suffix=None*)

Return the HTML DOM identifier for this field, e.g. something suitable to use as HTML input id. You can specify a suffix that will be included in the name when widget needs several inputs.

actual_fields(*form*)

Fields may be composed of other fields. For instance the `RichTextField` is containing a format field to define the text format. This method returns actual fields that should be considered for display / edition. It usually simply return self.

Form generation methods

form_init(form)

Method called at form initialization to trigger potential field initialization requiring the form instance. Do nothing by default.

typed_value(form, load_bytes=False)

Return the correctly typed value for this field in the form context.

Post handling methods**process_posted(form)**

Return an iterator on (field, value) that has been posted for field returned by [actual_fields\(\)](#).

process_form_value(form)

Return the correctly typed value posted for this field.

Now, you usually don't use that class but one of the concrete field classes described below, according to what you want to edit.

Basic fields**class cubicweb.web.formfields.StringField**

Use this field to edit unicode string (*String* yams type). This field additionally support a *max_length* attribute that specify a maximum size for the string (*None* meaning no limit).

Unless explicitly specified, the widget for this field will be:

- [Select](#) if some vocabulary is specified using *choices* attribute
- [TextInput](#) if maximum size is specified using *max_length* attribute and this length is inferior to 257.
- [TextArea](#) in all other cases

class cubicweb.web.formfields.PasswordField

Use this field to edit password (*Password* yams type, encoded python string).

Unless explicitly specified, the widget for this field will be a [PasswordInput](#).

class cubicweb.web.formfields.IntField

Use this field to edit integers (*Int* yams type). Similar to [BigIntField](#) but set max length when text input widget is used (the default).

class cubicweb.web.formfields.BigIntField

Use this field to edit big integers (*BigInt* yams type). This field additionally support *min* and *max* attributes that specify a minimum and/or maximum value for the integer (*None* meaning no boundary).

Unless explicitly specified, the widget for this field will be a [TextInput](#).

class cubicweb.web.formfields.FloatField

Use this field to edit floats (*Float* yams type). This field additionally support *min* and *max* attributes as the [IntField](#).

Unless explicitly specified, the widget for this field will be a [TextInput](#).

class cubicweb.web.formfields.BooleanField

Use this field to edit booleans (*Boolean* yams type).

Unless explicitly specified, the widget for this field will be a [Radio](#) with yes/no values. You can change that values by specifying *choices*.

class cubicweb.web.formfields.DateField

Use this field to edit date (*Date* yams type).

Unless explicitly specified, the widget for this field will be a *jQueryDatePicker*.

class cubicweb.web.formfields.DateTimeField

Use this field to edit datetime (*Datetime* yams type).

Unless explicitly specified, the widget for this field will be a *jQueryDateTimePicker*.

class cubicweb.web.formfields.TZDatetimeField

Use this field to edit a timezone-aware datetime (*TZDatetime* yams type). Note the posted values are interpreted as UTC, so you may need to convert them client-side, using some javascript in the corresponding widget.

class cubicweb.web.formfields.TimeField

Use this field to edit time (*Time* yams type).

Unless explicitly specified, the widget for this field will be a *jQueryTimePicker*.

class cubicweb.web.formfields.TimeIntervalField

Use this field to edit time interval (*Interval* yams type).

Unless explicitly specified, the widget for this field will be a *TextInput*.

Compound fields

class cubicweb.web.formfields.RichTextField

This compound field allow edition of text (unicode string) in a particular format. It has an inner field holding the text format, that can be specified using *format_field* argument. If not specified one will be automaticall generated.

Unless explicitly specified, the widget for this field will be a *FKEditor* or a *TextArea*. according to the field's format and to user's preferences.

class cubicweb.web.formfields.FileField

This compound field allow edition of binary stream (*Bytes* yams type). Three inner fields may be specified:

- *format_field*, holding the file's format.
- *encoding_field*, holding the file's content encoding.
- *name_field*, holding the file's name.

Unless explicitly specified, the widget for this field will be a *FileInput*. Inner fields, if any, will be added to a drop down menu at the right of the file input.

class cubicweb.web.formfields.CompoundField

This field shouldn't be used directly, it's designed to hold inner fields that should be conceptually grouped together.

Entity specific fields and function

class cubicweb.web.formfields.RelationField

Use this field to edit a relation of an entity.

Unless explicitly specified, the widget for this field will be a [Select](#).

cubicweb.web.formfields.guess_field(*eschema*, *rschema*, *role*='subject', *req*=None, ***kwargs*)

This function return the most adapted field to edit the given relation (*rschema*) where the given entity type (*eschema*) is the subject or object (*role*).

The field is initialized according to information found in the schema, though any value can be explicitly specified using *kwargs*.

Widgets

Note: A widget is responsible for the display of a field. It may use more than one HTML input tags. When the form is posted, a widget is also responsible to give back to the field something it can understand.

Of course you can not use any widget with any field...

class cubicweb.web.formwidgets.FieldWidget(*attrs*=None, *setdomid*=None, *suffix*=None)

The abstract base class for widgets.

Attributes

Here are standard attributes of a widget, that may be set on concrete class to override default behaviours:

needs_js list of javascript files needed by the widget.

needs_css list of css files needed by the widget.

setdomid flag telling if HTML DOM identifier should be set on input.

suffix string to use a suffix when generating input, to ease usage as a sub-widgets (eg widget used by another widget)

vocabulary_widget flag telling if this widget expect a vocabulary

Also, widget instances takes as first argument a *attrs* dictionary which will be stored in the attribute of the same name. It contains HTML attributes that should be set in the widget's input tag (though concrete classes may ignore it).

Form generation methods

render(*form*, *field*, *renderer*=None)

Called to render the widget for the given *field* in the given *form*. Return a unicode string containing the HTML snippet.

You will usually prefer to override the [_render\(\)](#) method so you don't have to handle addition of needed javascript / css files.

_render(*form*, *field*, *renderer*)

This is the method you have to implement in concrete widget classes.

values(*form, field*)

Return the current *string* values (i.e. for display in an HTML string) for the given field. This method returns a list of values since it's suitable for all kind of widgets, some of them taking multiple values, but you'll get a single value in the list in most cases.

Those values are searched in:

1. previously submitted form values if any (on validation error)
2. req.form (specified using request parameters)
3. extra form values given to form.render call (specified the code generating the form)
4. field's typed value (returned by its [typed_value\(\)](#) method)

Values found in 1. and 2. are expected to be already some 'display value' (eg a string) while those found in 3. and 4. are expected to be correctly typed value.

3 and 4 are handle by the `typed_value()` method to ease reuse in concrete classes.

attributes(*form, field*)

Return HTML attributes for the widget, automatically setting DOM identifier when desired (see `setdomid` attribute)

Post handling methods**process_field_data**(*form, field*)

Return process posted value(s) for widget and return something understandable by the associated *field*. That value may be correctly typed or a string that the field may parse.

HTML <input> based widgets

class cubicweb.web.formwidgets.**HiddenInput**(*attrs=None, setdomid=None, suffix=None*)

Simple <input type='hidden'> for hidden value, will return a unicode string.

class cubicweb.web.formwidgets.**TextInput**(*attrs=None, setdomid=None, suffix=None*)

Simple <input type='text'>, will return a unicode string.

class cubicweb.web.formwidgets.**EmailInput**(*attrs=None, setdomid=None, suffix=None*)

Simple <input type='email'>, will return a unicode string.

class cubicweb.web.formwidgets.**PasswordSingleInput**(*attrs=None, setdomid=None, suffix=None*)

Simple <input type='password'>, will return a utf-8 encoded string.

You may prefer using the [PasswordInput](#) widget which handles password confirmation.

class cubicweb.web.formwidgets.**FileInput**(*attrs=None, setdomid=None, suffix=None*)

Simple <input type='file'>, will return a tuple (name, stream) where name is the posted file name and stream a file like object containing the posted file data.

class cubicweb.web.formwidgets.**ButtonInput**(*attrs=None, setdomid=None, suffix=None*)

Simple <input type='button'>, will return a unicode string.

If you want a global form button, look at the [Button](#), [SubmitButton](#), [ResetButton](#) and [ImgButton](#) below.

Other standard HTML widgets

class cubicweb.web.formwidgets.**TextArea**(*attrs=None, setdomid=None, suffix=None*)

Simple <textarea>, will return a unicode string.

class cubicweb.web.formwidgets.**Select**(*attrs=None, multiple=False, **kwargs*)

Simple <select>, for field having a specific vocabulary. Will return a unicode string, or a list of unicode strings.

class cubicweb.web.formwidgets.**CheckBox**(*attrs=None, separator=None, **kwargs*)

Simple <input type='checkbox'>, for field having a specific vocabulary. One input will be generated for each possible value.

You can specify separator using the *separator* constructor argument, by default
 is used.

class cubicweb.web.formwidgets.**Radio**(*attrs=None, separator=None, **kwargs*)

Simple <input type='radio'>, for field having a specific vocabulary. One input will be generated for each possible value.

You can specify separator using the *separator* constructor argument, by default
 is used.

Date and time widgets

class cubicweb.web.formwidgets.**DateTimePicker**(*attrs=None, setdomid=None, suffix=None*)

<input type='text'> + javascript date/time picker for date or datetime fields. Will return the date or datetime as a unicode string.

class cubicweb.web.formwidgets.**JQueryDateTimePicker**(*initialtime=None, timesteps=15, separator=':', **kwargs*)

Compound widget using *JQueryDatePicker* and *JQueryTimePicker* widgets to define a date and time picker. Will return the date and time as python datetime instance.

class cubicweb.web.formwidgets.**JQueryDatePicker**(*datestr=None, min_of=None, max_of=None, **kwargs*)

Use jquery.ui.datepicker to define a date picker. Will return the date as a unicode string.

You can couple DatePickers by using the *min_of* and/or *max_of* parameters. The DatePicker identified by the value of *min_of*/(*max_of*) will force the user to choose a date anterior/(posterior) to this DatePicker.

example:

start and end are two JQueryDatePicker and start must always be before end:

```
affk.set_field_kwargs(etype, 'start_date', widget=JQueryDatePicker(min_of='end_date
↪'))
affk.set_field_kwargs(etype, 'end_date', widget=JQueryDatePicker(max_of='start_date
↪'))
```

That way, on change of end(/start) value a new max(/min) will be set for start(/end) The invalid dates will be gray colored in the datepicker

class cubicweb.web.formwidgets.**JQueryTimePicker**(*timestr=None, timesteps=30, separator=':', **kwargs*)

Use jquery.timePicker to define a time picker. Will return the time as a unicode string.

Ajax / javascript widgets

class cubicweb.web.formwidgets.**FCKEditor**(*args, **kwargs)

FCKEditor enabled <textarea>, will return a unicode string containing HTML formatted text.

class cubicweb.web.formwidgets.**AjaxWidget**(wdgtype, inputid=None, **kwargs)

Simple <div> based ajax widget, requiring a *wdgtype* argument telling which javascript widget should be used.

class cubicweb.web.formwidgets.**AutoCompletionWidget**(*args, **kwargs)

<input type='text'> based ajax widget, taking a *autocomplete_initfunc* argument which should specify the name of a method of the json controller. This method is expected to return allowed values for the input, that the widget will use to propose matching values as you type.

class cubicweb.web.formwidgets.**InOutWidget**(*args, **kwargs)

Other widgets

class cubicweb.web.formwidgets.**PasswordInput**(attrs=None, setdomid=None, suffix=None)

<input type='password'> and a confirmation input. Form processing will fail if password and confirmation differs, else it will return the password as a utf-8 encoded string.

class cubicweb.web.formwidgets.**IntervalWidget**(attrs=None, setdomid=None, suffix=None)

Custom widget to display an interval composed by 2 fields. This widget is expected to be used with a *CompoundField* containing the two actual fields.

Example usage:

```
class MyForm(FieldsForm):
    price = CompoundField(fields=(IntField(name='minprice'),
                                IntField(name='maxprice')),
                          label=_('price'),
                          widget=IntervalWidget())
```

class cubicweb.web.formwidgets.**BitSelect**(attrs=None, multiple=True, **kwargs)

Select widget for *IntField* using a vocabulary with bit masks as values.

See also *BitFieldFacet*.

class cubicweb.web.formwidgets.**HorizontalLayoutWidget**(attrs=None, setdomid=None, suffix=None)

Custom widget to display a set of fields grouped together horizontally in a form. See *IntervalWidget* for example usage.

class cubicweb.web.formwidgets.**EditableURLWidget**(attrs=None, setdomid=None, suffix=None)

Custom widget to edit separately a URL path / query string (used by default for the *path* attribute of *Bookmark* entities).

It deals with url quoting nicely so that the user edit the unquoted value.

Form controls

Those classes are not proper widget (they are not associated to field) but are used as form controls. Their API is similar to widgets except that *field* argument given to `render()` will be *None*.

```
class cubicweb.web.formwidgets.Button(label=('button_ok', 'OK_ICON'), attrs=None, setdomid=None,
                                       name="", value="", onclick=None, cwaction=None)
```

Simple `<input type='button'>`, base class for global form buttons.

Note that *label* is a msgid which will be translated at form generation time, you should not give an already translated string.

```
class cubicweb.web.formwidgets.SubmitButton(label=('button_ok', 'OK_ICON'), attrs=None,
                                             setdomid=None, name="", value="", onclick=None,
                                             cwaction=None)
```

Simple `<input type='submit'>`, main button to submit a form

```
class cubicweb.web.formwidgets.ResetButton(label=('button_ok', 'OK_ICON'), attrs=None,
                                             setdomid=None, name="", value="", onclick=None,
                                             cwaction=None)
```

Simple `<input type='reset'>`, main button to reset a form. You usually don't want to use this.

```
class cubicweb.web.formwidgets.ImgButton(domid, href, label, imgresource)
```

Simple `` wrapped into a `<a>` tag with href triggering something (usually a javascript call).

Base form classes

Besides the automatic form we'll see later, there are roughly two main form classes in *CubicWeb*:

```
class cubicweb.web.views.forms.FieldsForm(req, rset=None, row=None, col=None, submitmsg=None,
                                           mainform=True, **kwargs)
```

This is the base class for fields based forms.

Attributes

The following attributes may be either set on subclasses or given on form selection to customize the generated form:

needs_js sequence of javascript files that should be added to handle this form (through `add_js()`)

needs_css sequence of css files that should be added to handle this form (through `add_css()`)

domid value for the "id" attribute of the `<form>` tag

action value for the "action" attribute of the `<form>` tag

onsubmit value for the "onsubmit" attribute of the `<form>` tag

cssclass value for the "class" attribute of the `<form>` tag

cssstyle value for the "style" attribute of the `<form>` tag

cwtarget value for the "target" attribute of the `<form>` tag

redirect_path relative to redirect to after submitting the form

copy_nav_params flag telling if navigation parameters should be copied back in hidden inputs

form_buttons sequence of form control (*Button* widgets instances)

form_renderer_id identifier of the form renderer to use to render the form

fieldsets_in_order sequence of fieldset names , to control order

autocomplete set to False to add 'autocomplete=off' in the form open tag

Generic methods

Form.**field_by_name**(*name*, *role=None*)

Return field with the given name and role.

Raise `FieldNotFound` if the field can't be found.

Form.**fields_by_name**(*name*, *role=None*)

Return a list of fields with the given name and role.

Form construction methods

Form.**remove_field**(*field*)

Remove the given field.

Form.**append_field**(*field*)

Append the given field.

Form.**insert_field_before**(*field*, *name*, *role=None*)

Insert the given field before the field of given name and role.

Form.**insert_field_after**(*field*, *name*, *role=None*)

Insert the given field after the field of given name and role.

Form.**add_hidden**(*name*, *value=None*, ***kwargs*)

Append an hidden field to the form. *name*, *value* and extra keyword arguments will be given to the field constructor. The inserted field is returned.

Form rendering methods

render(*formvalues=None*, *renderer=None*, ***kwargs*)

Render this form, using the *renderer* given as argument or the default according to `form_renderer_id`. The rendered form is returned as a unicode string.

formvalues is an optional dictionary containing values that will be considered as field's value.

Extra keyword arguments will be given to renderer's `render()` method.

Form posting methods

Once a form is posted, you can retrieve the form on the controller side and use the following methods to ease processing. For "simple" forms, this should looks like :

```
form = self._cw.vreg['forms'].select('myformid', self._cw)
posted = form.process_posted()
# do something with the returned dictionary
```

Notice that form related to entity edition should usually use the *edit* controller which will handle all the logic for you.

process_posted()

use this method to process the content posted by a simple form. it will return a dictionary with field names as key and typed value as associated value.

iter_modified_fields(*editedfields=None*, *entity=None*)

return a generator on field that has been modified by the posted form.

```
class cubicweb.web.views.forms.EntityFieldsForm(_cw, rset=None, row=None, col=None, **kwargs)
```

This class is designed for forms used to edit some entities. It should handle for you all the underlying stuff necessary to properly work with the generic `EditController`.

As you have probably guessed, choosing between them is easy. Simply ask you the question ‘I am editing an entity or not?’. If the answer is yes, use [EntityFieldsForm](#), else use [FieldsForm](#).

Actually there exists a third form class:

```
class cubicweb.web.views.forms.CompositeForm(*args, **kwargs)
```

Form composed of sub-forms. Typical usage is edition of multiple entities at once.

but you’ll use this one rarely.

Renderers

Note: Form renderers are responsible to layout a form to HTML.

Here are the base renderers available:

```
class cubicweb.web.views.formrenderers.FormRenderer(req=None, rset=None, row=None, col=None, **kwargs)
```

This is the ‘default’ renderer, displaying fields in a two columns table:

field1 label	field1 input
field2 label	field2 input

buttons

```
class cubicweb.web.views.formrenderers.HTableFormRenderer(req=None, rset=None, row=None, col=None, **kwargs)
```

The ‘htable’ form renderer display fields horizontally in a table:

field1 label	field2 label	
field1 input	field2 input	buttons

```
class cubicweb.web.views.formrenderers.EntityCompositeFormRenderer(req=None, rset=None, row=None, col=None, **kwargs)
```

This is a specific renderer for the multiple entities edition form (‘muledit’).

Each entity form will be displayed in row off a table, with a check box for each entities to indicate which ones are edited. Those checkboxes should be automatically updated when something is edited.

```
class cubicweb.web.views.formrenderers.EntityFormRenderer(req=None, rset=None, row=None, col=None, **kwargs)
```

This is the ‘default’ renderer for entity’s form.

You can still use `form_renderer_id = ‘base’` if you want base `FormRenderer` layout even when selected for an entity.

```
class cubicweb.web.views.formrenderers.EntityInlinedFormRenderer(req=None, rset=None,
                                                                row=None, col=None,
                                                                **kwargs)
```

This is a specific renderer for entity's form inlined into another entity's form.

6.10.2 Dissection of an entity form

This is done (again) with a vanilla instance of the `tracker` cube. We will populate the database with a bunch of entities and see what kind of job the automatic entity form does.

Populating the database

We should start by setting up a bit of context: a project with two unpublished versions, and a ticket linked to the project and the first version.

```
>>> p = rql('INSERT Project P: P name "cubicweb"')
>>> for num in ('0.1.0', '0.2.0'):
...   rql('INSERT Version V: V num "%s", V version_of P WHERE P eid %(p)s' % num, {'p': p[0][0]})
...
<resultset 'INSERT Version V: V num "0.1.0", V version_of P WHERE P eid %(p)s' (1 rows):
↳ [765L] (('Version',))>
<resultset 'INSERT Version V: V num "0.2.0", V version_of P WHERE P eid %(p)s' (1 rows):
↳ [766L] (('Version',))>
>>> t = rql('INSERT Ticket T: T title "let us write more doc", T done_in V, '
            'T concerns P WHERE V num "0.1.0"', P eid %(p)s', {'p': p[0][0]})
>>> commit()
```

Now let's see what the edition form builds for us.

```
>>> cnx.use_web_compatible_requests('http://fakeurl.com')
>>> req = cnx.request()
>>> form = req.vreg['forms'].select('edition', req, rset=rql('Ticket T'))
>>> html = form.render()
```

Note: In order to play interactively with web side application objects, we have to cheat a bit to have request object that will looks like HTTP request object, by calling `use_web_compatible_requests()` on the connection.

This creates an automatic entity form. The `.render()` call yields an html (unicode) string. The html output is shown below (with internal fieldset omitted).

Looking at the html output

The form envelope

```
<div class="iformTitle"><span>main informations</span></div>
<div class="formBody">
  <form action="http://crater:9999/validateform" method="post" enctype="application/x-www-
  ↪form-urlencoded"
    id="entityForm" onsubmit="return freezeFormButtons(&#39;entityForm&#39;);"
    class="entityForm" target="eformframe">
    <div id="progress">validating...</div>
    <fieldset>
      <input name="__form_id" type="hidden" value="edition" />
      <input name="__errorurl" type="hidden" value="http://perdu.com#entityForm" />
      <input name="__domid" type="hidden" value="entityForm" />
      <input name="__type:763" type="hidden" value="Ticket" />
      <input name="eid" type="hidden" value="763" />
      <input name="__maineid" type="hidden" value="763" />
      <input name="_cw_edited_fields:763" type="hidden"
        value="concerns-subject,done_in-subject,priority-subject,type-subject,title-
        ↪subject,description-subject,__type,_cw_generic_field" />
      ...
    </fieldset>
    <iframe width="0px" height="0px" name="eformframe" id="eformframe" src="javascript:↪
    ↪void(0);"></iframe>
    </form>
  </div>
```

The main fieldset encloses a set of hidden fields containing various metadata, that will be used by the *edit controller* to process it back correctly.

The *freezeFormButtons(...)* javascript callback defined on the *onclick* event of the form element prevents accidental multiple clicks in a row.

The action of the form is mapped to the *validateform* controller (situated in *cubicweb.web.views.basecontrollers*).

A full explanation of the validation loop is given in *The form validation process*.

The attributes section

We can have a look at some of the inner nodes of the form. Some fields are omitted as they are redundant for our purposes.

```
<fieldset class="default">
  <table class="attributeForm">
    <tr class="title_subject_row">
      <th class="labelCol"><label class="required" for="title-subject:763">title</label>
      ↪</th>
      <td>
        <input id="title-subject:763" maxlength="128" name="title-subject:763" size="45"
          type="text" value="let us write more doc" />
      </td>
    </tr>
  </table>
</fieldset>
```

(continues on next page)

(continued from previous page)

```

</tr>
... (description field omitted) ...
<tr class="priority_subject_row">
  <th class="labelCol"><label class="required" for="priority-subject:763">priority</
↪label></th>
  <td>
    <select id="priority-subject:763" name="priority-subject:763" size="1">
      <option value="important">important</option>
      <option selected="selected" value="normal">normal</option>
      <option value="minor">minor</option>
    </select>
    <div class="helper">importance</div>
  </td>
</tr>
... (type field omitted) ...
<tr class="concerns_subject_row">
  <th class="labelCol"><label class="required" for="concerns-subject:763">concerns</
↪label></th>
  <td>
    <select id="concerns-subject:763" name="concerns-subject:763" size="1">
      <option selected="selected" value="760">Foo</option>
    </select>
  </td>
</tr>
<tr class="done_in_subject_row">
  <th class="labelCol"><label for="done_in-subject:763">done in</label></th>
  <td>
    <select id="done_in-subject:763" name="done_in-subject:763" size="1">
      <option value="__cubicweb_internal_field__"></option>
      <option selected="selected" value="761">Foo 0.1.0</option>
      <option value="762">Foo 0.2.0</option>
    </select>
    <div class="helper">version in which this ticket will be / has been done</div>
  </td>
</tr>
</table>
</fieldset>

```

Note that the whole form layout has been computed by the form renderer. It is the renderer which produces the table structure. Otherwise, the fields html structure is emitted by their associated widget.

While it is called the *attributes* section of the form, it actually contains attributes and *mandatory relations*. For each field, we observe:

- a dedicated row with a specific class, such as `title_subject_row` (responsability of the form renderer)
- an html widget (input, select, ...) with:
 - an id built from the `rtype-role:eid` pattern
 - a name built from the same pattern
 - possible values or preselected options

The relations section

```
<fieldset class="This ticket :">
  <legend>This ticket :</legend>
  <table class="attributeForm">
    <tr class="_cw_generic_field_None_row">
      <td colspan="2">
        <table id="relatedEntities">
          <tr><th>#160;</th><td>#160;</td></tr>
          <tr id="relationSelectorRow_763" class="separator">
            <th class="labelCol">
              <select id="relationSelector_763"
                onchange="javascript:showMatchingSelect(this.options[this.
↪selectedIndex].value,763);">
                <option value="">select a relation</option>
                <option value="appeared_in_subject">appeared in</option>
                <option value="custom_workflow_subject">custom workflow</option>
                <option value="depends_on_object">dependency of</option>
                <option value="depends_on_subject">depends on</option>
                <option value="identical_to_subject">identical to</option>
                <option value="see_also_subject">see also</option>
              </select>
            </th>
            <td id="unrelatedDivs_763"></td>
          </tr>
        </table>
      </td>
    </tr>
  </table>
</fieldset>
```

The optional relations are grouped into a drop-down combo box. Selection of an item triggers a javascript function which will:

- show already related entities in the div of id *relatedEntities* using a two-colown layout, with an action to allow deletion of individual relations (there are none in this example)
- provide a relation selector in the div of id *relationSelector_EID* to allow the user to set up relations and trigger dynamic action on the last div
- fill the div of id *unrelatedDivs_EID* with a dynamically computed selection widget allowing direct selection of an unrelated (but relatable) entity or a switch towards the *search mode* of *CubicWeb* which allows full browsing and selection of an entity using a dedicated action situated in the left column boxes.

The buttons zone

Finally comes the buttons zone.

```
<table width="100%">
  <tbody>
    <tr>
      <td align="center">
        <button class="validateButton" type="submit" value="validate">
```

(continues on next page)

(continued from previous page)

```

        
        validate
      </button>
    </td>
    <td style="align: right; width: 50%;">
      <button class="validateButton"
        onclick="postForm(&#39;__action_apply&#39;; , &#39;button_apply&#39;; , &#39;
↪entityForm&#39;);"
        type="button" value="apply">
        
        apply
      </button>
      <button class="validateButton"
        onclick="postForm(&#39;__action_cancel&#39;; , &#39;button_cancel&#39;; , &
↪#39;entityForm&#39;);"
        type="button" value="cancel">
        
        cancel
      </button>
    </td>
  </tr>
</tbody>
</table>

```

The most notable artifacts here are the `postForm(...)` calls defined on click events on these buttons. This function basically submits the form.

6.10.3 The form validation process

Validation loop

On form submission, the `form.action` is invoked. Basically, the `validateform` controller is called and its output lands in the specified target, an invisible `<iframe>` at the end of the form.

Hence, the main page is not replaced, only the `iframe` contents. The `validateform` controller only outputs a tiny javascript fragment which is then immediately executed.

```

<iframe width="0px" height="0px" name="eformframe" id="eformframe" src="javascript:↪
↪void(0);">
  <script type="text/javascript">
    window.parent.handleFormValidationResponse('entityForm', null, null,
        [false, [2164, {"name-subject": "required_
↪field"}]], null),
        null);
  </script>
</iframe>

```

The `window.parent` part ensures the javascript function is called on the right context (that is: the form element). We will describe its parameters:

- first comes the form id (*entityForm*)
- then two optional callbacks for the success and failure case
- an array containing:
 - a boolean which indicates status (success or failure), and then, on error:
 - * an array structured as `[eid, {'rtype-role': 'error msg'}, ...]`
 - on success:
 - * a url (string) representing the next thing to jump to

Given the array structure described above, it is quite simple to manipulate the DOM to show the errors at appropriate places.

Explanation

This mechanism may seem a bit overcomplicated but we have to deal with two realities:

- in the (strict) XHTML world, there are no iframes (hence the dynamic inclusion, tolerated by Firefox)
- no (or not all) browser(s) support file input field handling through ajax.

6.10.4 The *edit controller*

It can be found in (`cubicweb.web.views.editcontroller`). This controller processes data received from an html form to create or update entities.

Edition handling

The parameters related to entities to edit are specified as follows (first seen in *The attributes section*):

`<rtype-role>:<entity eid>`

where entity eid could be a letter in case of an entity to create. We name those parameters as *qualified*.

- Retrieval of entities to edit is done by using the forms parameters *eid* and *__type*
- For all the attributes and the relations of an entity to edit (attributes and relations are handled a bit differently but these details are not much relevant here) :
 - using the *rtype*, *role* and *__type* information, fetch an appropriate field instance
 - check if the field has been modified (if not, proceed to the next relation)
 - build an rql expression to update the entity

At the end, all rql expressions are executed.

- For each entity to edit:
 - if a qualified parameter *__linkto* is specified, its value has to be a string (or a list of strings) such as:

`<relation type>:<eids>:<target>`

where *<target>* is either *subject* or *object* and each eid could be separated from the others by a `_`. Target specifies if the *edited entity* is subject or object of the relation and each relation specified will be inserted.

- if a qualified parameter `__clone_eid` is specified for an entity, the relations of the specified entity passed as value of this parameter are copied on the edited entity.
- if a qualified parameter `__delete` is specified, its value must be a string or a list of string such as follows:

`<subjects eids>:<relation type>:<objects eids>`

where each eid subject or object can be separated from the other by `_`. Each specified relation will be deleted.

- If no entity is edited but the form contains the parameters `__linkto` and `eid`, this one is interpreted by using the value specified for `eid` to designate the entity on which to add the relations.

Note:

- if the parameter `__action_delete` is found, all the entities specified as to be edited will be deleted.
 - if the parameter `__action_cancel` is found, no action is completed.
 - if the parameter `__action_apply` is found, the editing is applied normally but the redirection is done on the form (see [Redirection control](#)).
 - if no entity is found to be edited and if there is no parameter `__action_delete`, `__action_cancel`, `__linkto`, `__delete` or `__insert`, an error is raised.
 - using the parameter `__message` in the form will allow to use its value as a message to provide the user once the editing is completed.
-

Redirection control

Once editing is completed, there is still an issue left: where should we go now? If nothing is specified, the controller will do his job but it does not mean we will be happy with the result. We can control that by using the following parameters:

- `__redirectpath`: path of the URL (relative to the root URL of the site, no form parameters)
- `__redirectparams`: forms parameters to add to the path
- `__redirectrql`: redirection RQL request
- `__redirectvid`: redirection view identifier
- `__errorurl`: initial form URL, used for redirecting in case a validation error is raised during editing. If this one is not specified, an error page is displayed instead of going back to the form (which is, if necessary, responsible for displaying the errors)
- `__form_id`: initial view form identifier, used if `__action_apply` is found

In general we use either `__redirectpath` and `__redirectparams` or `__redirectrql` and `__redirectvid`.

6.10.5 Examples

(Automatic) Entity form

Looking at some cubes available on the [cubicweb forge](#) we find some with form manipulation. The following example comes from the [conference](#) cube. It extends the change state form for the case where a Talk entity is getting into submitted state. The goal is to select reviewers for the submitted talk.

```
from cubicweb.web import formfields as ff, formwidgets as fwdgs
class SendToReviewerStatusChangeView(ChangeStateFormView):
    __select__ = (ChangeStateFormView.__select__ &
                  is_instance('Talk') &
                  rql_condition('X in_state S, S name "submitted"'))

    def get_form(self, entity, transition, **kwargs):
        form = super(SendToReviewerStatusChangeView, self).get_form(entity, transition,
→ **kwargs)
        relation = ff.RelationField(name='reviews', role='object',
                                   eidparam=True,
                                   label=_('select reviewers'),
                                   widget=fwdgs.Select(multiple=True))
        form.append_field(relation)
        return form
```

Simple extension of a form can be done from within the *FormView* wrapping the form. *FormView* instances have a handy `get_form` method that returns the form to be rendered. Here we add a *RelationField* to the base state change form.

One notable point is the `eidparam` argument: it tells both the field and the edit controller that the field is linked to a specific entity.

It is hence entirely possible to add ad-hoc fields that will be processed by some specialized instance of the edit controller.

Ad-hoc fields form

We want to define a form doing something else than editing an entity. The idea is to propose a form to send an email to entities in a resultset which implements *IEmailable*. Let's take a simplified version of what you'll find in `cubicweb.web.views.massmailing`.

Here is the source code:

```
def sender_value(form, field):
    return '%s <%s>' % (form._cw.user.dc_title(), form._cw.user.get_email())

def recipient_choices(form, field):
    return [(e.get_email(), e.eid)
            for e in form.cw_rset.entities()
            if e.get_email()]

def recipient_value(form, field):
    return [e.eid for e in form.cw_rset.entities()
            if e.get_email()]

class MassMailingForm(forms.FieldsForm):
```

(continues on next page)

(continued from previous page)

```

__regid__ = 'massmailing'

needs_js = ('cubicweb.widgets.js',)
domid = 'sendmail'
action = 'sendmail'

sender = ff.StringField(widget=TextInput({'disabled': 'disabled'}),
                        label=_('From:'),
                        value=sender_value)

recipient = ff.StringField(widget=CheckBox(),
                           label=_('Recipients:'),
                           choices=recipient_choices,
                           value=recipients_value)

subject = ff.StringField(label=_('Subject:'), max_length=256)

mailbody = ff.StringField(widget=AjaxWidget(wdgtype='TemplateTextField',
                                             inputid='mailbody'))

form_buttons = [ImgButton('sendbutton', "javascript: $('#sendmail').submit()",
                          _(_('send email'), 'SEND_EMAIL_ICON'),
                  ImgButton('cancelbutton', "javascript: history.back()",
                          stdmsgs.BUTTON_CANCEL, 'CANCEL_EMAIL_ICON')]

```

Let's detail what's going on up there. Our form will hold four fields:

- a sender field, which is disabled and will simply contains the user's name and email
- a recipients field, which will be displayed as a list of users in the context result set with checkboxes so user can still choose who will receive his mailing by checking or not the checkboxes. By default all of them will be checked since field's value return a list containing same eids as those returned by the vocabulary function.
- a subject field, limited to 256 characters (hence we know a *TextInput* will be used, as explained in *StringField*)
- a mailbody field. This field use an ajax widget, defined in *cubicweb.widgets.js*, and whose definition won't be shown here. Notice though that we tell this form need this javascript file by using *needs_js*

Last but not least, we add two buttons control: one to post the form using javascript (*\$('#sendmail')* being the jQuery call to get the element with DOM id set to 'sendmail', which is our form DOM id as specified by its *domid* attribute), another to cancel the form which will go back to the previous page using another javascript call. Also we specify an image to use as button icon as a resource identifier (see *Step 1: tired of the default look?*) given as last argument to *cubicweb.web.formwidgets.ImgButton*.

To see this form, we still have to wrap it in a view. This is pretty simple:

```

class MassMailingFormView(form.FormViewMixin, EntityView):
    __regid__ = 'massmailing'
    __select__ = is_instance(IEmailable) & authenticated_user()

    def call(self):
        form = self._cw.vreg['forms'].select('massmailing', self._cw,
                                             rset=self.cw_rset)

        form.render(w=self.w)

```

As you see, we simply define a view with proper selector so it only apply to a result set containing IEmailable entities, and so that only users in the managers or users group can use it. Then in the *call()* method for this view we simply select the above form and call its *.render()* method with our output stream as argument.

When this form is submitted, a controller with id ‘sendmail’ will be called (as specified using *action*). This controller will be responsible to actually send the mail to specified recipients.

Here is what it looks like:

```
class SendMailController(Controller):
    __regid__ = 'sendmail'
    __select__ = (authenticated_user() &
                  match_form_params('recipient', 'mailbody', 'subject'))

    def publish(self, rset=None):
        body = self._cw.form['mailbody']
        subject = self._cw.form['subject']
        eids = self._cw.form['recipient']
        # eids may be a string if only one recipient was specified
        if isinstance(eids, basestring):
            rset = self._cw.execute('Any X WHERE X eid %(x)s', {'x': eids})
        else:
            rset = self._cw.execute('Any X WHERE X eid in (%s)' % (','.join(eids)))
        recipients = list(rset.entities())
        msg = format_mail({'email' : self._cw.user.get_email(),
                          'name' : self._cw.user.dc_title()},
                          recipients, body, subject)
        if not self._cw.vreg.config.sendmails([(msg, recipients)]):
            msg = self._cw._('could not connect to the SMTP server')
        else:
            msg = self._cw._('emails successfully sent')
        raise Redirect(self._cw.build_url(__message=msg))
```

The entry point of a controller is the publish method. In that case we simply get back post values in request’s *form* attribute, get user instances according to eids found in the ‘recipient’ form value, and send email after calling *format_mail()* to get a proper email message. If we can’t send email or if we successfully sent email, we redirect to the index page with proper message to inform the user.

Also notice that our controller has a selector that deny access to it to anonymous users (we don’t want our instance to be used as a spam relay), but also checks if the expected parameters are specified in forms. That avoids later defensive programming (though it’s not enough to handle all possible error cases).

To conclude our example, suppose we wish a different form layout and that existent renderers are not satisfying (we would check that first of course :). We would then have to define our own renderer:

```
class MassMailingFormRenderer(formrenderers.FormRenderer):
    __regid__ = 'massmailing'

    def _render_fields(self, fields, w, form):
        w(u'<table class="headersform">')
        for field in fields:
            if field.name == 'mailbody':
                w(u'</table>')
                w(u'<div id="toolbar">')
                w(u'<ul>')
                for button in form.form_buttons:
```

(continues on next page)

(continued from previous page)

```

        w(u'<li>%s</li>' % button.render(form))
    w(u'</ul>')
    w(u'</div>')
    w(u'<div>')
    w(field.render(form, self))
    w(u'</div>')
else:
    w(u'<tr>')
    w(u'<td class="hlabel">%s</td>' %
        self.render_label(form, field))
    w(u'<td class="hvalue">')
    w(field.render(form, self))
    w(u'</td></tr>')

def render_buttons(self, w, form):
    pass

```

We simply override the `_render_fields` and `render_buttons` method of the base form renderer to arrange fields as we desire it: here we'll have first a two columns table with label and value of the sender, recipients and subject field (form order respected), then form controls, then a div containing the textarea for the email's content.

To bind this renderer to our form, we should add to our form definition above:

```
form_renderer_id = 'massmailing'
```

6.11 The facets system

Facets allow to restrict searches according to some user friendly criterias. CubicWeb has a builtin `facet` system to define restrictions `filters` really as easily as possible.

Here is an exemple of the facets rendering picked from our <http://www.cubicweb.org> web site:

**CubicWeb** CubicWeb

search

facets 

bookmark this search

has text

entity type

Blog entry

Tag

Ticket

in state

open

validation pending

done in

3.2.0

1.4.2

tagged by

OR 

cubes

facets

logilab

made with cubicweb

office search

portfolio

type

bug

story

created by

adimascio

alutz

nchauvat

sthenault

concerns

cubicweb

cubicweb-blog

Facets will appear on each page presenting more than one entity that may be filtered according to some known criteria.

6.11.1 Base classes for facets

The `cubicweb.web.facet` module contains a set of abstract classes to use as bases to build your own facets

All facet classes inherits from the `AbstractFacet` class, though you'll usually find some more handy class that do what you want.

Let's see available classes.

Classes you'll want to use

class `cubicweb.web.facet.RelationFacet`(*req*, *select=None*, *filtered_variable=None*, ***kwargs*)

Base facet to filter some entities according to other entities to which they are related. Create concrete facet by inheriting from this class and then configuring it by setting class attribute described below.

The relation is defined by the *rtype* and *role* attributes.

The *no_relation* boolean flag tells if a special 'no relation' value should be added (allowing to filter on entities which *do not* have the relation set). Default is computed according to the relation's cardinality.

The values displayed for related entities will be:

- result of calling their *label_vid* view if specified
- else their *target_attr* attribute value if specified
- else their *eid* (you usually want something nicer...)

When no *label_vid* is set, you will get translated value if *i18nable* is set. By default, *i18nable* will be set according to the schema, but you can force its value by setting it as a class attribute.

You can filter out target entity types by specifying *target_type*.

By default, vocabulary will be displayed sorted on *target_attr* value in an ascending way. You can control sorting with:

- *sortfunc*: set this to a stored procedure name if you want to sort on the result of this function's result instead of direct value
- *sortasc*: boolean flag to control ascendant/descendant sorting

To illustrate this facet, let's take for example an *excerpt* of the schema of an office location search application:

```
class Office(WorkflowableEntityType):
    price = Int(description='euros / m2 / HC / HT')
    surface = Int(description='m2')
    has_address = SubjectRelation('PostalAddress',
                                cardinality='1?',
                                composite='subject')
    proposed_by = SubjectRelation('Agency')
```

We can simply define a facet to filter offices according to the agency proposing it:

```
class AgencyFacet(RelationFacet):
    __regid__ = 'agency'
    # this facet should only be selected when visualizing offices
    __select__ = RelationFacet.__select__ & is_instance('Office')
```

(continues on next page)

(continued from previous page)

```
# this facet is a filter on the 'Agency' entities linked to the office
# through the 'proposed_by' relation, where the office is the subject
# of the relation
rtype = 'has_address'
# 'subject' is the default but setting it explicitly doesn't hurt...
role = 'subject'
# we want to display the agency's name
target_attr = 'name'
```

class cubicweb.web.facet.RelationAttributeFacet(req, select=None, filtered_variable=None, **kwargs)

Base facet to filter some entities according to an attribute of other entities to which they are related. Most things work similarly as [RelationFacet](#), except that:

- *label_vid* doesn't make sense here
- you should specify the attribute type using *target_attr_type* if it's not a String
- you can specify a comparison operator using *comparator*

Back to our example... if you want to search office by postal code and that you use a [RelationFacet](#) for that, you won't get the expected behaviour: if two offices have the same postal code, they've however two different addresses. So you'll see in the facet the same postal code twice, though linked to a different address entity. There is a great chance your users won't understand that...

That's where this class come in! It's used to said that you want to filter according to the *attribute value* of a related entity, not to the entity itself. Now here is the source code for the facet:

```
class PostalCodeFacet(RelationAttributeFacet):
    __regid__ = 'postalcode'
    # this facet should only be selected when visualizing offices
    __select__ = RelationAttributeFacet.__select__ & is_instance('Office')
    # this facet is a filter on the PostalAddress entities linked to the
    # office through the 'has_address' relation, where the office is the
    # subject of the relation
    rtype = 'has_address'
    role = 'subject'
    # we want to search according to address 'postal_code' attribute
    target_attr = 'postalcode'
```

class cubicweb.web.facet.HasRelationFacet(req, select=None, filtered_variable=None, **kwargs)

This class simply filter according to the presence of a relation (whatever the entity at the other end). It display a simple checkbox that lets you refine your selection in order to get only entities that actually have this relation. You simply have to define which relation using the *rtype* and *role* attributes.

Here is an example of the rendering of thos facet to filter book with image and the corresponding code:



```
class HasImageFacet(HasRelationFacet):
    __regid__ = 'hasimage'
    __select__ = HasRelationFacet.__select__ & is_instance('Book')
    rtype = 'has_image'
    role = 'subject'
```



```
class cubicweb.web.facet.AttributeFacet(req, select=None, filtered_variable=None, **kwargs)
```

Base facet to filter some entities according one of their attribute. Configuration is mostly similarly as *RelationAttributeFacet*, except that:

- *target_attr* doesn't make sense here (you specify the attribute using *rtype*)
- *role* neither, it's systematically 'subject'

So, suppose that in our office search example you want to refine search according to the office's surface. Here is a code snippet achieving this:

```
class SurfaceFacet(AttributeFacet):
    __regid__ = 'surface'
    __select__ = AttributeFacet.__select__ & is_instance('Office')
    # this facet is a filter on the office's surface
    rtype = 'surface'
    # override the default value of operator since we want to filter
    # according to a minimal value, not an exact one
    comparator = '>='

    def vocabulary(self):
        """override the default vocabulary method since we want to
        hard-code our threshold values.

        Not overriding would generate a filter containing all existing
        surfaces defined in the database.
        """
        return [('> 200', '200'), ('> 250', '250'),
                ('> 275', '275'), ('> 300', '300')]
```

```
class cubicweb.web.facet.RQLPathFacet(*args, **kwargs)
```

Base facet to filter some entities according to an arbitrary rql path. Path should be specified as a list of 3-uples or triplet string, where 'X' represent the filtered variable. You should specify using *filter_variable* the snippet variable that will be used to filter out results. You may also specify a *label_variable*. If you want to filter on an attribute value, you usually don't want to specify the later since it's the same as the filter variable, though you may have to specify the attribute type using *restr_attr_type* if there are some type ambiguity in the schema for the attribute.

Using this facet, we can rewrite facets we defined previously:

```
class AgencyFacet(RQLPathFacet):
    __regid__ = 'agency'
    # this facet should only be selected when visualizing offices
    __select__ = is_instance('Office')
    # this facet is a filter on the 'Agency' entities linked to the office
    # through the 'proposed_by' relation, where the office is the subject
    # of the relation
    path = ['X has_address O', 'O name N']
    filter_variable = 'O'
    label_variable = 'N'

class PostalCodeFacet(RQLPathFacet):
    __regid__ = 'postalcode'
    # this facet should only be selected when visualizing offices
    __select__ = is_instance('Office')
```

(continues on next page)

(continued from previous page)

```
# this facet is a filter on the PostalAddress entities linked to the
# office through the 'has_address' relation, where the office is the
# subject of the relation
path = ['X has_address O', 'O postal_code PC']
filter_variable = 'PC'
```

Though some features, such as ‘no value’ or automatic internationalization, won’t work. This facet class is designed to be used for cases where *RelationFacet* or *RelationAttributeFacet* can’t do the trick (e.g. when you want to filter on entities where are not directly linked to the filtered entities).

class cubicweb.web.facet.*RangeFacet*(req, select=None, filtered_variable=None, **kwargs)

This class allows to filter entities according to an attribute of numerical type.

It displays a slider using *jquery* to choose a lower bound and an upper bound.

The example below provides an alternative to the surface facet seen earlier, in a more powerful way since

- lower/upper boundaries are computed according to entities to filter
- user can specify lower/upper boundaries, not only the lower one

```
class SurfaceFacet(RangeFacet):
    __regid__ = 'surface'
    __select__ = RangeFacet.__select__ & is_instance('Office')
    # this facet is a filter on the office's surface
    rtype = 'surface'
```

All this with even less code!

The image below display the rendering of the slider:



class cubicweb.web.facet.*DateRangeFacet*(req, select=None, filtered_variable=None, **kwargs)

This class works similarly as the *RangeFacet* but for attribute of date type.

The image below display the rendering of the slider for a date range:



class cubicweb.web.facet.*BitFieldFacet*(req, select=None, filtered_variable=None, **kwargs)

Base facet class for Int field holding some bit values using binary masks.

label / value for each bit should be given using the choices attribute.

See also *BitSelect*.

class cubicweb.web.facet.*AbstractRangeRQLPathFacet*(*args, **kwargs)

The *AbstractRangeRQLPathFacet* is the base class for *RQLPathFacet*-type facets allowing the use of *RangeWidgets*-like widgets (such as (*FacetRangeWidget*, class:*DateFacetRangeWidget*) on the parent *RQLPathFacet* target attribute.

class cubicweb.web.facet.**RangeRQLPathFacet**(*args, **kwargs)

The *RangeRQLPathFacet* uses the FacetRangeWidget on the *AbstractRangeRQLPathFacet* target attribute

class cubicweb.web.facet.**DateRangeRQLPathFacet**(*args, **kwargs)

The *DateRangeRQLPathFacet* uses the DateFacetRangeWidget on the *AbstractRangeRQLPathFacet* target attribute

Classes for facets implementor

Unless you didn't find the class that does the job you want above, you may want to skip those classes...

class cubicweb.web.facet.**AbstractFacet**(req, select=None, filtered_variable=None, **kwargs)

Abstract base class for all facets. Facets are stored in their own 'facets' registry. They are similar to contextual components since they use the following configurable properties:

- *visible*, boolean flag telling if a facet should be displayed or not
- *order*, integer to control facets display order
- *context*, telling if a facet should be displayed in the table form filter (context = 'tablefilter') or in the facet box (context = 'facetbox') or in both (context = '')

The following methods define the facet API:

get_widget()

Return the widget instance to use to display this facet, or None if the facet can't do anything valuable (only one value in the vocabulary for instance).

add_rql_restrictions()

When some facet criteria has been updated, this method is called to add restriction for this facet into the rql syntax tree. It should get back its value in form parameters, and modify the syntax tree (*self.select*) accordingly.

Facets will have the following attributes set (beside the standard *AppObject* ones):

- *select*, the *rql.stmts.Select* node of the rql syntax tree being filtered
- *filtered_variable*, the variable node in this rql syntax tree that we're interested in filtering

Facets implementors may also be interested in the following properties / methods:

operator

Return the operator (AND or OR) to use for this facet when multiple values are selected.

rqlexec(rql, args=None)

Utility method to execute some rql queries, and simply returning an empty list if *Unauthorized* is raised.

class cubicweb.web.facet.**VocabularyFacet**(req, select=None, filtered_variable=None, **kwargs)

This abstract class extends *AbstractFacet* to use the *FacetVocabularyWidget* as widget, suitable for facets that may restrict values according to a (usually computed) vocabulary.

A class which inherits from *VocabularyFacet* must define at least these methods:

vocabulary()

Return vocabulary for this facet, eg a list of 2-uple (label, value).

possible_values()

Return a list of possible values (as string since it's used to compare to a form value in javascript) for this facet.

6.12 Internationalization

Cubicweb fully supports the internalization of its content and interface.

Cubicweb's interface internationalization is based on the translation project [GNU gettext](#).

Cubicweb' internalization involves two steps:

- in your Python code and cubicweb-tal templates : mark translatable strings
- in your instance : handle the translation catalog, edit translations

6.12.1 String internationalization

User defined string

In the Python code and cubicweb-tal templates translatable strings can be marked in one of the following ways :

- by using the *built-in* function `_`:

```
class PrimaryView(EntityView):
    """the full view of an non final entity"""
    __regid__ = 'primary'
    title = _('primary')
```

OR

- by using the equivalent request's method:

```
class NoResultView(View):
    """default view when no result has been found"""
    __regid__ = 'noresult'

    def call(self, **kwargs):
        self.w(u'<div class="searchMessage"><strong>%s</strong></div>\n'
              % self._cw._('No result matching query'))
```

The goal of the *built-in* function `_` is only **to mark the translatable strings**, it will only return the string to translate itself, but not its translation (it's actually another name for the *unicode* builtin).

In the other hand the request's method `self._cw._` is also meant to retrieve the proper translation of translation strings in the requested language.

Finally you can also use the `__` (two underscores) attribute of request object to get a translation for a string *which should not itself added to the catalog*, usually in case where the actual msgid is created by string interpolation

```
self._cw.___('This %s' % etype)
```

In this example `._cw.__` is used instead of `._cw._` so we don't have 'This %s' in messages catalogs.

Translations in cubicweb-tal template can also be done with TAL tags *i18n:content* and *i18n:replace*.

If you need to mark other messages as translatable, you can create a file named *i18n/static-messages.pot*, see for example [Specialize translation for an application cube](#).

You could put there messages not found in the python sources or overrides some messages that are in cubes used in the dependencies.

Generated string

We do not need to mark the translation strings of entities/relations used by a particular instance's schema as they are generated automatically. String for various actions are also generated.

For example the following schema:

```
class EntityA(EntityType):
    relation_a2b = SubjectRelation('EntityB')

class EntityB(EntityType):
    pass
```

May generate the following message

```
add EntityA relation_a2b EntityB subject
```

This message will be used in views of EntityA for creation of a new EntityB with a preset relation relation_a2b between the current EntityA and the new EntityB. The opposite message

```
add EntityA relation_a2b EntityB object
```

Is used for similar creation of an EntityA from a view of EntityB. The title of they respective creation form will be

```
creating EntityB (EntityA %(linkto)s relation_a2b EntityB)

creating EntityA (EntityA relation_a2b %(linkto)s EntityA)
```

In the translated string you can use %(linkto)s for reference to the source entity.

6.12.2 Handling the translation catalog

Once the internationalization is done in your code, you need to populate and update the translation catalog. Cubicweb provides the following commands for this purpose:

- *i18ncubicweb* updates Cubicweb framework's translation catalogs. Unless you actually work on the framework itself, you don't need to use this command.
- *i18ncube* updates the translation catalogs of *one particular cube* (or of all cubes). After this command is executed you must update the translation files *.po* in the "i18n" directory of your cube. This command will of course not remove existing translations still in use. It will mark unused translation but not remove them.
- *i18ninstance* recompiles the translation catalogs of *one particular instance* (or of all instances) after the translation catalogs of its cubes have been updated. This command is automatically called every time you create or update your instance. The compiled catalogs (*.mo*) are stored in the *i18n/<lang>/LC_MESSAGES* of instance where *lang* is the language identifier ('en' or 'fr' for example).

Example

You have added and/or modified some translation strings in your cube (after creating a new view or modifying the cube's schema for example). To update the translation catalogs you need to do:

1. `cubicweb-ctl i18ncube <cube>`
2. Edit the `<cube>/i18n/xxx.po` files and add missing translations (those with an empty `msgstr`)
3. `hg ci -m "updated i18n catalogs"`
4. `cubicweb-ctl i18ninstance <myinstance>`

6.12.3 Customizing the messages extraction process

The messages extraction performed by the `i18ncommand` collects messages from a few different sources:

- the schema and application definition (entity names, docstrings, help messages, `uicfg`),
- the source files:
 - `i18n:content` or `i18n:replace` directives from TAL files (with `.pt` extension),
 - strings prefixed by an underscore (`_`) in python files,
 - strings with double quotes prefixed by an underscore in javascript files.

The source files are collected by walking through the cube directory, but ignoring a few directories like `.hg`, `.tox`, `test` or `node_modules`.

If you need to customize this behaviour in your cube, you have to extend the `cubicweb.devtools.devctl.I18nCubeMessageExtractor`. The example below will collect strings from `jinja2` files and ignore the `static` directory during the messages collection phase:

```
# mymodule.py
from cubicweb.devtools import devctl

class MyMessageExtractor(devctl.I18nCubeMessageExtractor):

    blacklist = devctl.I18nCubeMessageExtractor | {'static'}
    formats = devctl.I18nCubeMessageExtractor.formats + ['jinja2']

    def collect_jinja2(self):
        return self.find('.jinja2')

    def extract_jinja2(self, files):
        return self._gettext(files, output='jinja.pot',
                              extraopts='-L python --from-code=utf-8')
```

Then, you'll have to register it with a `cubicweb.i18ncube` entry point in your cube's `setup.py`:

```
setup(
    # ...
    entry_points={
        # ...
        'cubicweb.i18ncube': [
            'mycube=cubicweb_mycube.mymodule:MyMessageExtractor',
        ],
    },
)
```

(continues on next page)

(continued from previous page)

```

    },
    # ...
)

```

6.12.4 Editing po files

Using a PO aware editor

Many tools exist to help maintain .po (PO) files. Common editors or development environment provides modes for these. One can also find dedicated PO files editor, such as [poedit](#).

While usage of such a tool is commendable, PO files are perfectly editable with a (unicode aware) plain text editor. It is also useful to know their structure for troubleshooting purposes.

Structure of a PO file

In this section, we selectively quote passages of the [GNU gettext](#) manual chapter on PO files, available there:

```
https://www.gnu.org/software/hello/manual/gettext/PO-Files.html
```

One PO file entry has the following schematic structure:

```

white-space
# translator-comments
#. extracted-comments
#: reference...
#, flag...
#| msgid previous-untranslated-string
msgid untranslated-string
msgstr translated-string

```

A simple entry can look like this:

```

#: lib/error.c:116
msgid "Unknown system error"
msgstr "Error desconegut del sistema"

```

It is also possible to have entries with a context specifier. They look like this:

```

white-space
# translator-comments
#. extracted-comments
#: reference...
#, flag...
#| msgctxt previous-context
#| msgid previous-untranslated-string
msgctxt context
msgid untranslated-string
msgstr translated-string

```

The context serves to disambiguate messages with the same untranslated-string. It is possible to have several entries with the same untranslated-string in a PO file, provided that they each have a different context. Note that an empty context string and an absent msgctxt line do not mean the same thing.

Contexts and CubicWeb

CubicWeb PO files have both non-contextual and contextual msgids.

Contextual entries are automatically used in some cases. For instance, `entity.dc_type()`, `eschema.display_name(req)` or `display_name(etype, req, form, context)` methods/function calls will use them.

It is also possible to explicitly use a context with `_cw.pgettext(context, msgid)`.

Specialize translation for an application cube

Every cube has its own translation files. For a specific application cube it can be useful to specialize translations of other cubes. You can either mark those strings for translation using `_` in the python code, or add a *static-messages.pot* file into the *i18n* directory. This file looks like:

```
msgid ""
msgstr ""
"PO-Revision-Date: YEAR-MO-DA HO:MI +ZONE\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Generated-By: pygettext.py 1.5\n"
"Plural-Forms: nplurals=2; plural=(n > 1);\n"

msgig "expression to be translated"
msgstr ""
```

Doing this, `expression to be translated` will be taken into account by the `i18ncube` command and additional messages will then appear in *.po* files of the cube.

6.13 The property mechanism

6.13.1 Property API

6.13.2 Registering and using your own property

6.14 HTTP cache management

HTTP cache managers

6.14.1 Cache policies

class cubicweb.web.httppcache.**NoHTTPCacheManager**(*view*)

default cache manager: set no-cache cache control policy

class cubicweb.web.httppcache.**MaxAgeHTTPCacheManager**(*view*)

max-age cache manager: set max-age cache control policy, with max-age specified with the *cache_max_age* attribute of the view

class cubicweb.web.httppcache.**EtagHTTPCacheManager**(*view*)

etag based cache manager for startup views

- etag is generated using the view name and the user's groups
- set policy to 'must-revalidate' and expires to the current time to force revalidation on each request

class cubicweb.web.httppcache.**EntityHTTPCacheManager**(*view*)

etag based cache manager for view displaying a single entity

- etag is generated using entity's eid, the view name and the user's groups
- get last modified time from the entity definition (this may not be the entity's modification time since a view may include some related entities with a modification time to consider) using the *last_modified* method

6.14.2 Exception

exception cubicweb.web.httppcache.**NoEtag**

an etag can't be generated

6.14.3 Helper functions

cubicweb.web.httppcache.**set_http_cache_headers**(*self*)

6.15 Locate resources

WebConfiguration.**locate_resource**(*rid*)

return the (directory, filename) where the given resource may be found

WebConfiguration.**locate_doc_file**(*fname*)

return the directory where the given resource may be found

WebConfiguration.**locate_all_files**(*rid*, *rdirectory*='wdoc')

return all files corresponding to the given resource

6.16 Static files handling

`WebConfiguration.static_directory`

`WebConfiguration.static_file_exists(rpath)`

`WebConfiguration.static_file_open(rpath, mode='wb')`

`WebConfiguration.static_file_add(rpath, data)`

`WebConfiguration.static_file_del(rpath)`

PYRAMID

`cubicweb.pyramid` provides a way to bind a CubicWeb data repository to a Pyramid WSGI web application.

It can be used in two different ways:

- Through the `pyramid` *command* or through `cubicweb.pyramid.wsgi_application()` WSGI application factory, one can run an **all-in-one** CubicWeb instance with the web part served by a Pyramid application. This is referred to as the *backwards compatible mode*.
- Through the `pyramid` configuration type, one can setup a CubicWeb instance which repository can be used from within a Pyramid application. Such an instance may be launched through `pserve` or any WSGI server as would any plain Pyramid application.

7.1 Quick start

7.1.1 Prerequisites

Install the *pyramid* flavour of CubicWeb (here with `pip`, possibly in a `virtualenv`):

```
pip install cubicweb
```

7.1.2 Instance creation and running

In *backwards compatible mode*

In this mode, you can simply create an instance of kind **all-in-one** with the `cubicweb-ctl create` command. You'll then need to add a `pyramid.ini` file in your instance directory, see *Pyramid Settings file* for details about the content of this file.

Start the instance with the '*pyramid*' *command* instead of 'start':

```
cubicweb-ctl pyramid --debug myinstance
```

Without *backwards compatibility*

In this mode, you can create an instance of kind `pyramid` as follow:

```
cubicweb-ctl create -c pyramid <cube_name> <instance_name>
```

This will bootstrap a `development.ini` file typical of a Pyramid application in the instance's directory. The new instance may then be launched by any WSGI server, for instance with `pserve`:

```
pserve etc/cubicweb.d/<instance_name>/development.ini
```

In a pyramid application

- Create a pyramid application
- Include `cubicweb.pyramid`:

```
def includeme(config):  
    # ...  
    config.include('cubicweb.pyramid')  
    # ...
```

- Configure the instance name (in the `.ini` file):

```
cubicweb.instance = myinstance
```

- Configure the base-url in `all-in-one.conf` to match the ones of the pyramid configuration (this is a temporary limitation).

7.2 The ‘pyramid’ command

The ‘pyramid’ command is a replacement for the ‘start’ command of *cubicweb-ctl tool*. It provides the same options and a few other ones.

7.2.1 Options

--no-daemon

Run the server in the foreground.

--debug-mode

Activate the repository debug mode (logs in the console and the debug toolbar). Implies *--no-daemon*.

Also force the following pyramid options:

```
pyramid.debug_authorization = yes  
pyramid.debug_notfound = yes  
pyramid.debug_routematch = yes  
pyramid.reload_templates = yes
```

-D, --debug

Equals to *--debug-mode --no-daemon --reload*

--reload

Restart the server if any source file is changed

--reload-interval=RELOAD_INTERVAL

Interval, in seconds, between file modifications checks [current: 1]

-l <log level>, --loglevel=<log level>

Set the loglevel. debug if -D is set, error otherwise

--profile-output=PROFILE_OUTPUT

Profiling output file (default: "program.prof")

--profile-dump-every=N

Dump profile stats to output every N requests (default: 100)

7.3 Settings

7.3.1 Cubicweb Settings

Pyramid CubicWeb will **not** make use of the configuration entries found in the cubicweb configuration (a.k.a. *all-in-one.conf*) for any pyramid related configuration value.

7.3.2 Pyramid Settings file

In *backwards compatibility* mode, Pyramid settings will be looked for in a `pyramid.ini` file in the instance home directory (where the `all-in-one.conf` file is), its `[main]` section will be read and used as the settings of the pyramid Configurator.

This configuration file is almost the same as the one read by `pserve`, which allow to easily add any pyramid extension and configure it.

A typical `pyramid.ini` file is:

```
[main]
pyramid.includes =
    pyramid_redis_sessions

cubicweb.defaults = no
cubicweb.includes =
    cubicweb.pyramid.auth
    cubicweb.pyramid.login

cubicweb.profile = no

redis.sessions.secret = your_cookie_signing_secret
redis.sessions.timeout = 1200

redis.sessions.host = mywheezy
```

Without *backwards compatibility* a standard `development.ini` file can be used with any useful CubicWeb-specific settings added.

7.3.3 Pyramid CubicWeb configuration entries

The Pyramid CubicWeb specific configuration entries are:

cubicweb.instance (string)

A CubicWeb instance name. Useful when the application is not run by *The ‘pyramid’ command*.

cubicweb.debug (bool)

Enables the cubicweb debugmode. Works only if the instance is setup by `cubicweb.instance`.

Unlike when the debugmode is set by the `cubicweb-ctl pyramid --debug-mode` command, the pyramid debug options are untouched.

cubicweb.includes (list)

Same as `pyramid.includes`, but the includes are done after the cubicweb specific registry entries are initialized.

Useful to include extensions that requires these entries.

cubicweb.bwcompat (bool)

(True) Enable/disable backward compatibility. This only applies to “all-in-one” configuration type.

See `cubicweb.pyramid.bwcompat`.

cubicweb.bwcompat.errorhandler (bool)

(True) Enable/disable the backward compatibility error handler. Set to ‘no’ if you need to define your own error handlers.

cubicweb.defaults (bool)

(True) Enable/disable defaults. See `cubicweb.pyramid.defaults`.

cubicweb.auth.update_login_time (bool)

(True) Add a `cubicweb.pyramid.auth.UpdateLoginTimeAuthenticationPolicy` policy, that update the `CWUser.login_time` attribute when a user login.

cubicweb.auth.authtgt (bool)

(True) Enables the 2 cookie-base auth policies, which activate/deactivate depending on the *persistent* argument passed to *remember*.

The default login views set persistent to True if a `__setauthcookie` parameters is passed to them, and evals to True in `pyramid.settings.asbool()`.

The configuration values of the policies are arguments for `pyramid.authentication.AuthTktAuthenticationPolicy`.

The first policy handles session authentication. It doesn’t get activated if *remember()* is called with *persistent=False*:

cubicweb.auth.authtgt.session.cookie_name (str)

(‘auth_tkt’) The cookie name. Must be different from the persistent authentication cookie name.

cubicweb.auth.authtgt.session.samesite (str)

(‘auth_tkt’) Allows you to declare if your cookie should be restricted to a first-party or same-site context. See [here](#) for more information.

cubicweb.auth.authtgt.session.timeout (int)

(1200) Cookie timeout.

cubicweb.auth.auth_tkt.session.reissue_time (int)

(120) Reissue time.

The second policy handles persistent authentication. It doesn't get activated if *remember()* is called with *persistent=True*:

cubicweb.auth.auth_tkt.persistent.cookie_name (str)

('auth_tkt') The cookie name. Must be different from the session authentication cookie name.

cubicweb.auth.auth_tkt.persistent.samesite (str)

('auth_tkt') Allows you to declare if your cookie should be restricted to a first-party or same-site context. See [here](#) for more information.

cubicweb.auth.auth_tkt.persistent.max_age (int)

(30 days) Max age in seconds.

cubicweb.auth.auth_tkt.persistent.reissue_time (int)

(1 day) Reissue time in seconds.

Both policies set the `secure` flag to `True` by default, meaning that cookies will only be sent back over a secure connection (see [Authentication Policies documentation](#) for details). This can be configured through `cubicweb.auth.auth_tkt.persistent.secure` and `cubicweb.auth.auth_tkt.session.secure` configuration options.

cubicweb.auth.groups_principals (bool)

(True) Setup a callback on the authentication stack that inject the user groups in the principals.

7.4 Authentication

7.4.1 Overview

A default authentication stack is provided by the `cubicweb.pyramid.auth` module, which is included by `cubicweb.pyramid.default`.

The authentication stack is built around `pyramid_multiauth`, and provides a few default policies that reproduce the default cubicweb behavior.

Note: Note that this module only provides an authentication policy, not the views that handle the login form. See [cubicweb.pyramid.login](#)

7.4.2 Customize

The default policies can be individually deactivated, as well as the default authentication callback that returns the current user groups as principals.

The following settings can be set to *False*:

- `cubicweb.auth.update_login_time`. Activate the policy that update the user *login_time* when *remember* is called.
- `cubicweb.auth.auth_tkt` and all its subvalues.
- `cubicweb.auth.groups_principals`

Additional policies can be added by accessing the `MultiAuthenticationPolicy` instance in the registry:

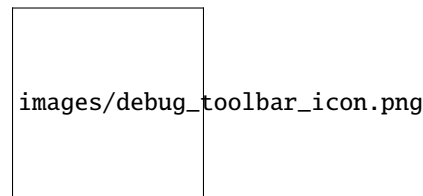
```
mypolicy = SomePolicy()
authpolicy = config.registry['cubicweb.authpolicy']
authpolicy._policies.append(mypolicy)
```

7.5 The pyramid debug toolbar

The pyramid webserver comes with an integrated [debug toolbar](#) that offers a lot of information to ease development. To ease the development process in CubicWeb a series of custom debug panels have been developed especially for that purpose.

To use the pyramid debug toolbar in CubicWeb, you need to:

- install it either by doing a `pip install pyramid_debugtoolbar` or following [the official installation instructions](#)
- launch the pyramid command adding the `-t/--toolbar` argument to enable it like so: `cubicweb-ctl pyramid my_instance -t` (you probably want to add `-D` to activate the debug mode during development)
- navigate to the website and click on the icon on the right like on this screenshot:



And you'll have access to the debug toolbar content for this page.

7.5.1 Custom panels

A series of custom debug panels have been written to offer more useful debug information during development. Here is the list:

General 'CubicWeb' Panel

Provides:

- currently selected controller for this with and uri/requests information
- current instance configuration, options that differs from default ones are in bold
- a list of useful links like on the default CW home

Screenshot:

CubicWeb general panel

Controller

Kind	Request	Path	Controller
view	<cubicweb.pyramid.core.CubicWebPyramidRequest object at 0x7f43061abd68> [source]	/	<cubicweb.web.views.basecontrollers.ViewController object at 0x7f43061b25c0> [source]

Configuration

Key	Value	Default	Help
access-control-allow-headers	[]	()	comma-separated list of HTTP headers the application may set in the response
access-control-allow-methods	[]	()	comma-separated list of allowed HTTP methods
access-control-allow-origin	[]	()	comma-separated list of allowed origin domains or "" for any domain
access-control-expose-headers	[]	()	comma-separated list of HTTP headers the application declare in response to a preflight request
access-control-max-age	None	None	maximum age of cross-origin resource sharing (in seconds)
allow-email-login	False	False	allow users to login with their primary email if set
anonymize-jsonp-queries	True	True	anonymize the connection before executing any jsonp query.
anonymous-password	anon	None	password of the CubicWeb user account to use for anonymous user, if anonymous-user is set
anonymous-user	anon	None	login of the CubicWeb user account to use for anonymous user (if you want to allow anonymous)
auth-mode	cookie	cookie	authentication mode (cookie / http)
base-url	http://incubus.in.fpsource.info:8080/	None	web server root url

Registry Decisions Panel

Provides:

- a list of all decisions taken in all registry during this page construction
- the arguments given to take the decision
- all the selection entities during decisions with their score
- which one has won if any

Registry Decisions

Result	Decision
'controllers' -> cubicweb.web.views.basecontrollers.ViewController [source]	
End score: 0.5 args: (<cubicweb.pyramid.core.CubicWebPyramidRequest object at 0x7f43061abd68>,) kwargs: <ul style="list-style-type: none">• 'appli': <cubicweb.web.application.CubicWebPublisher object at 0x7f430d68e860>	<ul style="list-style-type: none">• 0.5: cubicweb.web.views.basecontrollers.ViewController [source]
'views' -> cubicweb.web.views.startup.IndexView [source]	
End score: 1 args: (<cubicweb.pyramid.core.CubicWebPyramidRequest object at 0x7f43061abd68>,) kwargs: <ul style="list-style-type: none">• 'rset': None	<ul style="list-style-type: none">• 1: cubicweb.web.views.startup.IndexView [source]
'views' -> cubicweb.web.views.basetemplates.TheMainTemplate [source]	
End score: True args: (<cubicweb.pyramid.core.CubicWebPyramidRequest object at 0x7f43061abd68>,) kwargs: <ul style="list-style-type: none">• 'rset': None• 'view': <cubicweb.web.views.startup.IndexView object at 0x7f43061b2470>	<ul style="list-style-type: none">• 0: cubicweb.web.views.basetemplates.NonTemplatableViewTemplate [source]• 0: cubicweb.web.views.basetemplates.ModalMainTemplate [source]• True: cubicweb.web.views.basetemplates.TheMainTemplate [source]
'views' -> cubicweb_bootstrap.views.BSHTMLHeader [source]	
End score: 0.5 args: (<cubicweb.pyramid.core.CubicWebPyramidRequest object at 0x7f43061abd68>,) kwargs: <ul style="list-style-type: none">• 'rset': None	<ul style="list-style-type: none">• 0.5: cubicweb_bootstrap.views.BSHTMLHeader [source]

Registry Store

Provides:

- a listing of all the content of the different registries
- for each entity its detailed information

Registry Store

actions

adapters

after_add_entity_hooks

after_add_relation_hooks

after_delete_entity_hooks

after_delete_relation_hooks

after_update_entity_hooks

ajax-func

before_add_entity_hooks

before_add_relation_hooks

before_delete_entity_hooks

before_delete_relation_hooks

before_update_entity_hooks

components

controllers

ctxcomponents

etypes

facets

Actions

about	<ul style="list-style-type: none"> • cubicweb.web.views.wdoc.AboutAction [source] <ul style="list-style-type: none"> ◦ regid: 'about' ◦ select: 'yes' ◦ select name: 'yes' ◦ select score: '0.5' ◦ registries: ('actions',)
addentity	<ul style="list-style-type: none"> • cubicweb.web.views.actions.AddNewAction [source] <ul style="list-style-type: none"> ◦ regid: 'addentity' ◦ select: 'AndPredicate(match_search_state(normal),OrPredicate(addable_etype_empty_rset,AndPredicate(multi_lines_rset,one_etype_rset,has_add_per ◦ select name: 'AndPredicate' ◦ registries: ('actions',)
addrelated	<ul style="list-style-type: none"> • cubicweb.web.views.actions.AddRelatedActions [source] <ul style="list-style-type: none"> ◦ regid: 'addrelated' ◦ select: 'AndPredicate(match_search_state(normal),one_line_rset,non_final_entity)' ◦ select name: 'AndPredicate' ◦ registries: ('actions',)
cancel	<ul style="list-style-type: none"> • cubicweb.web.views.actions.CancelSelectAction [source] <ul style="list-style-type: none"> ◦ regid: 'cancel' ◦ select: 'match_search_state(linksearch)' ◦ select name: 'match_search_state' ◦ registries: ('actions',)
copy	<ul style="list-style-type: none"> • cubicweb.web.views.actions.CopyAction [source] <ul style="list-style-type: none"> ◦ regid: 'copy' ◦ select: 'AndPredicate(match_search_state(normal),one_line_rset,has_permission)' ◦ select name: 'AndPredicate' ◦ registries: ('actions',)
cw.source-sync	<ul style="list-style-type: none"> • cubicweb.web.views.cwsources.CWSourceSyncAction [source] <ul style="list-style-type: none"> ◦ regid: 'cw.source-sync'

RQL

Provides:

- a list of all executed RQL queries during this page creation
- for each RQL query all the generated SQL queries
- detail information like the result, the args and the description of each query
- the call stack on each query to see where it has been called

RQL queries

#	Time (ms)	RQL	Result	SQL	Description	Stack
1	7.40	Any GN WHERE U in_group G, G name GN, U [['guests']] eid %(userid)s { 'userid': 816 }		SELECT _G.cw_name FROM cw_CWGroup AS _G, in_group_relation AS rel_in_group0 WHERE rel_in_group0.eid_from=% (userid)s AND rel_in_group0.eid_to=_G.cw_eid { 'userid': 816 }	<cubicweb.utils.RepeatList at 139925841979208 item=('String',) size=1>	show stack
2	0.57	Any K, V WHERE P for_user U, U eid %(userid)s, P pkey K, P value V { 'userid': 816 }	[]	SELECT _P.cw_pkey, _P.cw_value FROM cw_CWProperty AS _P WHERE _P.cw_for_user=(userid)s { 'userid': 816 }	<cubicweb.utils.RepeatList at 139925841980944 item=('String', 'String') size=0>	show stack
3	0.82	Any X WHERE EXISTS(X identity U, X eid %(x)s, U eid %(u)s) { 'x': 816, 'u': 816 }	[[816]]	SELECT %(x)s WHERE EXISTS(SELECT 1 WHERE %(x)s=% (u)s) { 'x': 816, 'u': 816 }	<cubicweb.utils.RepeatList at 139925841980888 item=('CWUser',) size=1>	show stack
4	1.09	Any B,T,P ORDERBY lower(T) WHERE B is Bookmark,B title T, B path P, B bookmarked_by U, U eid %(x)s { 'x': 816 }	[]	SELECT _B.cw_eid, _B.cw_title, _B.cw_path FROM bookmarked_by_relation AS rel_bookmarked_by0, cw_Bookmark AS _B WHERE rel_bookmarked_by0.eid_from=_B.cw_eid AND rel_bookmarked_by0.eid_to=%(x)s ORDER BY LOWER(_B.cw_title) { 'x': 816 }	<cubicweb.utils.RepeatList at 139925842100800 item=('Bookmark', 'String', 'String') size=0>	show stack

RQL queries

#	Time (ms)	RQL	Result	SQL	Description	Stack
1	7.40	Any GN WHERE U in_group G, G name GN, U [['guests']] eid %(userid)s { 'userid': 816 }		SELECT _G.cw_name FROM cw_CWGroup AS _G, in_group_relation AS rel_in_group0 WHERE rel_in_group0.eid_from=% (userid)s AND rel_in_group0.eid_to=_G.cw_eid { 'userid': 816 }	<cubicweb.utils.RepeatList at 139925841979208 item=('String',) size=1>	hide stack
<pre> 1 File "/home/psychojoker/.pythonz/python3/CPython-3.7.3/lib/python3.7/threading.py", line 885, in _bootstrap 2 self._bootstrap_inner() 3 File "/home/psychojoker/.pythonz/python3/CPython-3.7.3/lib/python3.7/threading.py", line 917, in _bootstrap_inner 4 self.run() 5 File "/home/psychojoker/.pythonz/python3/CPython-3.7.3/lib/python3.7/threading.py", line 865, in run 6 self.target(*self.args, **self.kwargs) 7 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/task.py", line 85, in handler_thread 8 task.service() 9 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/channel.py", line 356, in service 10 task.service() 11 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/task.py", line 172, in service 12 self.execute() 13 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/task.py", line 440, in execute 14 app_iter = self.channel.server.application(environs, start_response) 15 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/proxy_headers.py", line 62, in translate_proxy_headers 16 return app(environs, start_response) 17 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/wsgicors.py", line 204, in __call__ 18 return self.application(environs, custom_start_response) 19 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/pyramid/router.py", line 270, in __call__ 20 response = self.execution_policy(environs, self) 21 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/pyramid/router.py", line 277, in default_execution_policy 22 return router.invoke_request(request) 23 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/pyramid/router.py", line 249, in invoke_request 24 response = handle_request(request) 25 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/pyramid_debugtoolbar/toolbar.py", line 257, in toolbar_tween 26 response = handler(request) 27 File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/pyramid_debugtoolbar/panels/performance.py", line 58, in resource_timer_handler 28 result = handler(request) </pre>						

SQL

Provides:

- a list of all executed SQL queries during this page creation
- for each SQL query the RQL query that has generated it, if any (some aren't)
- detail information like the result, the args and if the query has rollback
- the call stack on each query to see where it has been called

SQL queries

#	Time (ms)	SQL	Rollback? From RQL	Stack
1	6.73	<pre>SELECT _G.cw_name FROM cw_CWGroup AS _G, in_group_relation AS rel_in_group0 WHERE rel_in_group0.eid_from=%(userid)s AND rel_in_group0.eid_to=_G.cw_eid {'userid': 816}</pre>	False Any GN WHERE U in_group G, G name GN, U eid %(userid)s	show stack
2	0.12	<pre>SELECT _P.cw_pkey, _P.cw_value FROM cw_CWProperty AS _P WHERE _P.cw_for_user=%(userid)s {'userid': 816}</pre>	False Any K, V WHERE P for_user U, U eid %(userid)s, P pkey K, P value V	show stack
3	0.09	<pre>SELECT %(x)s WHERE EXISTS(SELECT 1 WHERE %(x)s=%(u)s) {'x': 816, 'u': 816}</pre>	False Any X WHERE EXISTS(X identity U, X eid %(x)s, U eid %(u)s)	show stack
4	0.41	<pre>SELECT _B.cw_eid, _B.cw_title, _B.cw_path FROM bookmarked_by relation AS rel_bookmarked_by0, cw_Bookmark AS _B WHERE rel_bookmarked_by0.eid_from=_B.cw_eid AND rel_bookmarked_by0.eid_to=%(x)s ORDER BY LOWER(_B.cw_title) {'x': 816}</pre>	False Any B,T,P ORDERBY lower(T) WHERE B is Bookmark,B title T, B path P, B bookmarked_by U, U eid %(x)s	show stack
5	0.57	<pre>SELECT %(o)s, %(u)s WHERE EXISTS(SELECT 1 FROM cw_CWGroup AS _G, in_group_relation AS rel_in_group0 WHERE %(o)s=%(u)s AND rel_in_group0.eid_from=%(u)s AND rel_in_group0.eid_to=_G.cw_eid AND _G.cw_name=%(139925841214728)s) {'o': 816, 'u': 816, '139925841214728': 'users'}</pre>	False Any O,U WHERE EXISTS(O identity U, U in_group G, G name "users", O eid %(o)s, U eid %(u)s)	show stack
6	0.43	<pre>SELECT COUNT(_X.cw_eid) FROM cw_Blog AS _X {}</pre>	False Any COUNT(X) WHERE X is Blog	show stack

7.5.2 Accessing the sources of the class/functions/method listing the debug panels

A traversal of all those custom panels is the see the source code of all listing class/functions/methods. You can access those by:

- clicking on the `[source]` close to the target when available
- clicking on the file path in the traceback stack

Registry Decisions

Result	Decision
'controllers' -> cubicweb.web.views.basecontrollers.ViewController [source]	
End score: 0.5 args: (<cubicweb.pyramid.core.CubicWebPyramidRequest object at 0x7f43061abd68>,) kwargs: <ul style="list-style-type: none"> • 'appli': <cubicweb.web.application.CubicWebPublisher object at 0x7f430d68e860> 	<ul style="list-style-type: none"> • 0.5: cubicweb.web.views.basecontrollers.ViewController [source]
'views' -> cubicweb.web.views.startup.IndexView [source]	
End score: 1 args: (<cubicweb.pyramid.core.CubicWebPyramidRequest object at 0x7f43061abd68>,) kwargs: <ul style="list-style-type: none"> • 'rset': None 	<ul style="list-style-type: none"> • 1: cubicweb.web.views.startup.IndexView [source]

#	Time (ms)	SQL	Rollback?	From RQL
1	6.73	<pre>SELECT G.cw_name FROM cw.CWGroup AS G, in_group_relation AS rel_in_group0 WHERE rel_in_group0.eid_from=%(userid)s AND rel_in_group0.eid_to=G.cw_eid {'userid': 816}</pre>	False	Any GN WHERE U in_group G, G name GN, U eid %(userid)s
1		File "/home/psychojoker/.pythonz/pythons/CPython-3.7.3/lib/python3.7/threading.py", line 887, in _bootstrap		
2		self._bootstrap_inner()		
3		File "/home/psychojoker/.pythonz/pythons/CPython-3.7.3/lib/python3.7/threading.py", line 917, in _bootstrap_inner		
4		self.run()		
5		File "/home/psychojoker/.pythonz/pythons/CPython-3.7.3/lib/python3.7/threading.py", line 865, in run		
6		self._target(*self._args, **self._kwargs)		
7		File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/task.py", line 85, in handler_thread		
8		task.service()		
9		File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/channel.py", line 356, in service		
10		task.service()		
11		File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/task.py", line 172, in service		
12		self.execute()		
13		File "/home/psychojoker/code/yunohost/ynh-dev/logilab/cubicweb/ve3/lib/python3.7/site-packages/waitress/task.py", line 440, in execute		
14		app_iter = self.channel.server.application(environs, start_response)		

Link to source code

You be sent to a page looking like this:

```
190
197 class DeleteAction(action.Action):
198     __regid__ = 'delete'
199     __select__ = action.Action.__select__ & has_permission('delete')
200
201     title = _('delete')
202     category = 'moreactions'
203     order = 20
204
205     def url(self):
206         if len(self.cw_rset) == 1:
207             entity = self.cw_rset.get_entity(self.cw_row or 0, self.cw_col or 0)
208             return self.cw.build_url(entity.rest_path(), vid='deleteconf')
209         return self.cw.build_url(rql=self.cw_rset.printable_rql(), vid='deleteconf')
210
211
212 class CopyAction(action.Action):
213     __regid__ = 'copy'
214     __select__ = (action.Action.__select__ & one_line_rset()
215                  & has_permission('add'))
216
217     title = _('copy')
218     category = 'moreactions'
219     order = 30
220
221     def url(self):
222         entity = self.cw_rset.get_entity(self.cw_row or 0, self.cw_col or 0)
223         return entity.absolute_url(vid='copy')
224
225
226 class AddNewAction(MultipleEditAction):
227     """when we're seeing more than one entity with the same type, propose to
228     add a new one
229     """
230     __regid__ = 'addentity'
231     __select__ = (action.Action.__select__ &
232                  (addable_etype_empty_rset()
233                   | (multi_lines_rset() & one_etype_rset() & has_add_permission())))
234
235
236     category = 'moreactions'
```

7.5.3 Contributing

All source code for the custom panels is located [here](#) and the documentation of how to write custom toolbar panels [here](#).

ADDITIONAL SERVICES

In this chapter, we introduce services crossing the *web - repository - administration* organisation of the first parts of the CubicWeb book. Those services can be either proper services (like the undo functionality) or mere *topical cross-sections* across CubicWeb.

8.1 Undoing changes in CubicWeb

Many desktop applications offer the possibility for the user to undo its last changes : this *undo feature* has now been integrated into the CubicWeb framework. This document will introduce you to the *undo feature* both from the end-user and the application developer point of view.

But because a semantic web application and a common desktop application are not the same thing at all, especially as far as undoing is concerned, we will first introduce *what* is the *undo feature* for now.

8.1.1 What's *undoing* in a CubicWeb application

What is an *undo feature* is quite intuitive in the context of a desktop application. But it is a bit subtler in the context of a Semantic Web application. This section introduces some of the main differences between a classical desktop and a Semantic Web applications to keep in mind in order to state precisely *what we want*.

The notion transactions

A CubicWeb application acts upon an *Entity-Relationship* model, described by a schema. This allows to ensure some data integrity properties. It also implies that changes are made by all-or-none groups called *transactions*, such that the data integrity is preserved whether the transaction is completely applied *or* none of it is applied.

A transaction can thus include more actions than just those directly required by the main purpose of the user. For example, when a user *just* writes a new blog entry, the underlying *transaction* holds several *actions* as illustrated below :

- By admin on 2012/02/17 15:18 - Created Blog entry : Torototo
 1. Created Blog entry : Torototo
 2. Added relation : Torototo owned by admin
 3. Added relation : Torototo blog entry of Undo Blog
 4. Added relation : Torototo in state draft (draft)
 5. Added relation : Torototo created by admin

Because of the very nature (all-or-none) of the transactions, the “undoable stuff” are the transactions and not the actions !

Public and private actions within a transaction

Actually, within the *transaction* “Created Blog entry : Torototo”, two of those *actions* are said to be *public* and the others are said to be *private*. *Public* here means that the public actions (1 and 3) were directly requested by the end user ; whereas *private* means that the other actions (2, 4, 5) were triggered “under the hood” to fulfill various requirements for the user operation (ensuring integrity, security, ...).

And because quite a lot of actions can be triggered by a “simple” end-user request, most of which the end-user is not (and does not need or wish to be) aware, only the so-called public actions will appear¹ in the description of the an undoable transaction.

- By admin on 2012/02/17 15:18 - Created Blog entry : Torototo

1. Created Blog entry : Torototo
2. Added relation : Torototo blog entry of Undo Blog

But note that both public and private actions will be undone together when the transaction is undone.

(In)dependent transactions : the simple case

A CubicWeb application can be used *simultaneously* by different users (whereas a single user works on an given office document at a given time), so that there is not always a single history time-line in the CubicWeb case. Moreover CubicWeb provides security through the mechanism of *permissions* granted to each user. This can lead to some transactions *not* being undoable in some contexts.

In the simple case two (unprivileged) users Alice and Bob make relatively independent changes : then both Alice and Bob can undo their changes. But in some case there is a clean dependency between Alice’s and Bob’s actions or between actions of one of them. For example let’s suppose that :

- Alice has created a blog,
- then has published a first post inside,
- then Bob has published a second post in the same blog,
- and finally Alice has updated its post contents.

Then it is clear that Alice can undo her contents changes and Bob can undo his post creation independently. But Alice can not undo her post creation while she has not first undone her changes. It is also clear that Bob should *not* have the permissions to undo any of Alice’s transactions.

More complex dependencies between transactions

But more surprising things can quickly happen. Going back to the previous example, Alice *can* undo the creation of the blog after Bob has published its post in it ! But this is possible only because the schema does not *require* for a post to be in a blog. Would the *blog entry of* relation have been mandatory, then Alice could not have undone the blog creation because it would have broken integrity constraint for Bob’s post.

When a user attempts to undo a transaction the system will check whether a later transaction has explicit dependency on the would-be-undone transaction. In this case the system will not even attempt the undo operation and inform the user.

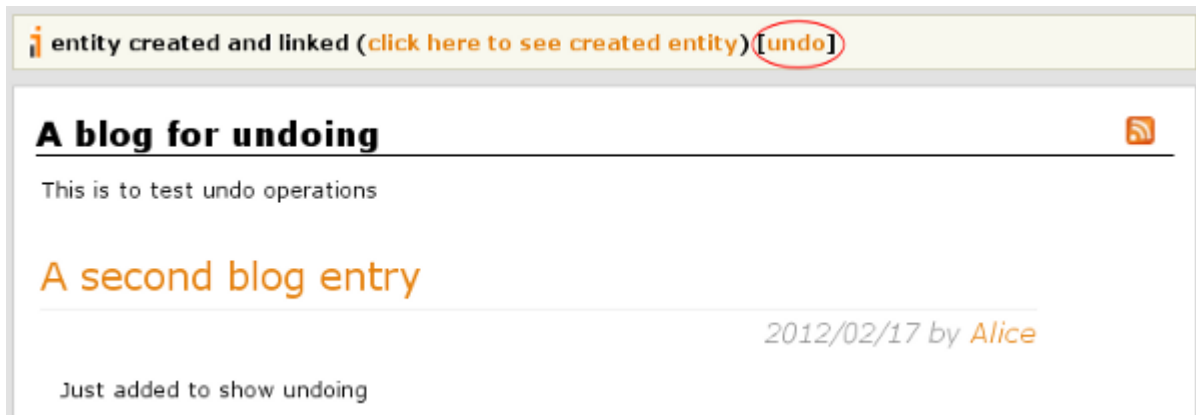
¹ The end-user Web interface could be improved to enable user to choose whether he wishes to see private actions.

If no such dependency is detected the system will attempt the undo operation but it can fail, typically because of integrity constraint violations. In such a case the undo operation is completely³ rolledback.

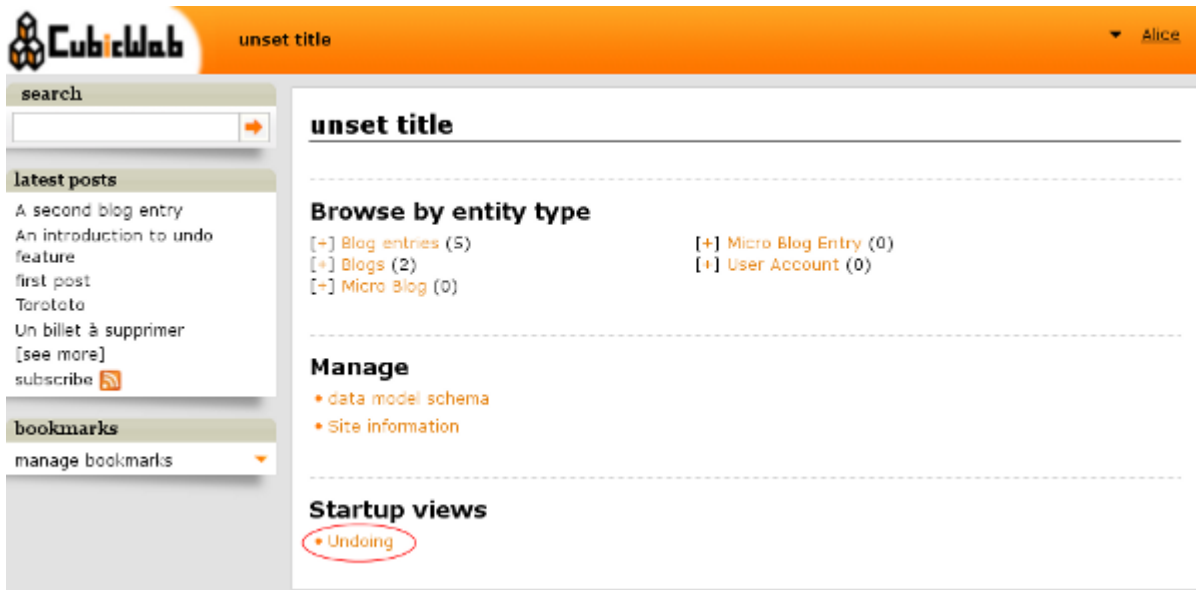
8.1.2 The *undo* feature for CubicWeb end-users

The exposition of the undo feature to the end-user through a Web interface is still quite basic and will be improved toward a greater usability. But it is already fully functional. For now there are two ways to access the *undo feature* as long as the it has been activated in the instance configuration file with the option *undo-support=yes*.

Immediately after having done the change to be canceled through the **undo** link in the message. This allows to undo an hastily action immediately. For example, just after having validated the creation of the blog entry *A second blog entry* we get the following message, allowing to undo the creation.



At any time we can access the **undo-history view** accessible from the start-up page.



This view will provide inspection of the transaction and their (public) actions. Each transaction provides its own **undo** link. Only the transactions the user has permissions to see and undo will be shown.

³ Meaning none of the actions in the transaction is undone. Depending upon the application, it might make sense to enable *partial* undo. That is to say undo in which some actions could not be undo without preventing to undo the others actions in the transaction (as long as it does not break schema integrity). This is not forbidden by the back-end but is deliberately not supported by the front-end (for now at least).



If the user attempts to undo a transaction which can't be undone or whose undoing fails, then a message will explain the situation and no partial undoing will be left behind.

This is all for the end-user side of the undo mechanism : this is quite simple indeed ! Now, in the following section, we are going to introduce the developer side of the undo mechanism.

8.1.3 The *undo* feature for CubicWeb application developers

A word of warning : this section is intended for developers, already having some knowledge of what's under CubicWeb's hood. If it is not *yet* the case, please refer to CubicWeb documentation <http://docs.cubicweb.org/> .

Overview

The core of the undo mechanisms is at work in the *native source*, beyond the RQL. This does mean that *transactions* and *actions* are *no entities*. Instead they are represented at the SQL level and exposed through the *DB-API* supported by the repository *Connection* objects.

Once the *undo feature* has been activated in the instance configuration file with the option *undo-support=yes*, each mutating operation (cf.²) will be recorded in some special SQL table along with its associated transaction. Transaction are identified by a *txuuid* through which the functions of the *DB-API* handle them.

On the web side the last committed transaction *txuuid* is remembered in the request's data to allow for immediate undoing whereas the *undo-history* view relies upon the *DB-API* to list the accessible transactions. The actual undoing is performed by the *UndoController* accessible at URL of the form *www.my.host/my/instance/undo?txuuid=...*

² There is only five kind of elementary actions (beyond merely accessing data for reading):

- **C** : creating an entity
- **D** : deleting an entity
- **U** : updating an entity attributes
- **A** : adding a relation
- **R** : removing a relation

The repository side

Please refer to the file *cubicweb/server/sources/native.py* and *cubicweb/transaction.py* for the details.

The undoing information is mainly stored in three SQL tables:

transactions Stores the txuuid, the user eid and the date-and-time of the transaction. This table is referenced by the two others.

tx_entity_actions Stores the undo information for actions on entities.

tx_relation_actions Stores the undo information for the actions on relations.

When the undo support is activated, entries are added to those tables for each mutating operation on the data repository, and are deleted on each transaction undoing.

Those table are accessible through the following methods of the repository *Connection* object :

undoable_transactions Returns a list of *Transaction* objects accessible to the user and according to the specified filter(s) if any.

tx_info Returns a *Transaction* object from a *txuuid*

undo_transaction Returns the list of *Action* object for the given *txuuid*.

NB: By default it only return *public* actions.

The web side

The exposure of the *undo feature* to the end-user through the Web interface relies on the *DB-API* introduced above. This implies that the *transactions* and *actions* are not *entities* linked by *relations* on which the usual views can be applied directly.

That's why the file *cubicweb/web/views/undohistory.py* defines some dedicated views to access the undo information :

UndoHistoryView This is a *StartupView*, the one accessible from the home page of the instance which list all transactions.

UndoableTransactionView This view handles the display of a single *Transaction* object.

UndoableActionBaseView This (abstract) base class provides private methods to build the display of actions whatever their nature.

Undoable[Add|Remove|Create|Delete|Update]ActionView Those views all inherit from *UndoableActionBaseView* and each handles a specific kind of action.

UndoableActionPredicate This predicate is used as a *selector* to pick the appropriate view for actions.

Apart from this main *undo-history view* a *txuuid* is stored in the request's data *last_undoable_transaction* in order to allow immediate undoing of a hastily validated operation. This is handled in *cubicweb/web/application.py* in the *main_publish* and *add_undo_link_to_msg* methods for the storing and displaying respectively.

Once the undo information is accessible, typically through a *txuuid* in an *undo* URL, the actual undo operation can be performed by the *UndoController* defined in *cubicweb/web/views/basecontrollers.py*. This controller basically extracts the *txuuid* and performs a call to *undo_transaction* and in case of an undo-specific error, lets the top level publisher handle it as a validation error.

8.1.4 Conclusion

The undo mechanism relies upon a low level recording of the mutating operation on the repository. Those records are accessible through some method added to the *DB-API* and exposed to the end-user either through a whole history view or through an immediate undoing link in the message box.

The undo feature is functional but the interface and configuration options are still quite reduced. One major improvement would be to be able to filter with a finer grain which transactions or actions one wants to see in the *undo-history view*. Another critical improvement would be to enable the undo feature on a part only of the entity-relationship schema to avoid storing too much useless data and reduce the underlying overhead.

But both functionality are related to the strong design choice not to represent transactions and actions as entities and relations. This has huge benefits in terms of safety and conceptual simplicity but prevents from using lots of convenient CubicWeb features such as *facets* to access undo information.

Before developing further the undo feature or eventually revising this design choice, it appears that some return of experience is strongly needed. So don't hesitate to try the undo feature in your application and send us some feedback.

8.1.5 Notes

APPENDIXES

The following chapters are reference material.

9.1 Frequently Asked Questions (FAQ)

9.1.1 Generalities

Why do you use the LGPL license to prevent me from doing X ?

LGPL means that *if* you redistribute your application, you need to redistribute the changes you made to CubicWeb under the LGPL licence.

Publishing a web site has nothing to do with redistributing source code according to the terms of the LGPL. A fair amount of companies use modified LGPL code for internal use. And someone could publish a *CubicWeb* component under a BSD licence for others to plug into a LGPL framework without any problem. The only thing we are trying to prevent here is someone taking the framework and packaging it as closed source to his own clients.

Why does not CubicWeb have a template language ?

There are enough template languages out there. You can use your preferred template language if you want.

CubicWeb does not define its own templating language as this was not our goal. Based on our experience, we realized that we could gain productivity by letting designers use design tools and developpers develop without the use of the templating language as an intermediary that could not be anyway efficient for both parties. Python is the templating language that we use in *CubicWeb*, but again, it does not prevent you from using a templating language.

Moreover, CubicWeb currently supports `simpletag` out of the box and it is also possible to use the `cwtags` library to build html trees using the `with statement` with more comfort than raw strings.

Why do you think using pure python is better than using a template language ?

Python is an Object Oriented Programming language and as such it already provides a consistent and strong architecture and syntax a templating language would not reach.

Using Python instead of a template language for describing the user interface makes it to maintain with real functions/classes/contexts without the need of learning a new dialect. By using Python, we use standard OOP techniques and this is a key factor in a robust application.

CubicWeb looks pretty recent. Is it stable ?

It is constantly evolving, piece by piece. The framework has evolved since 2001 and data has been migrated from one schema to the other ever since. There is a well-defined way to handle data and schema migration.

You can see the roadmap there: <https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/boards>.

Why is the RQL query language looking similar to X ?

It may remind you of SQL but it is higher level than SQL, more like SPARQL. Except that SPARQL did not exist when we started the project. With version 3.4, CubicWeb has support for SPARQL.

The RQL language is what is going to make a difference with django- like frameworks for several reasons.

1. accessing data is *much* easier with it. One can write complex queries with RQL that would be tedious to define and hard to maintain using an object/filter suite of method calls.
2. it offers an abstraction layer allowing your applications to run on multiple back-ends. That means not only various SQL backends (postgresql, sqlite, sqlserver, mysql), but also non-SQL data stores like LDAP directories and subversion/mercurial repositories (see the *vcfile* component).

Which ajax library is CubicWeb using ?

CubicWeb uses [jQuery](#) and provides a few helpers on top of that. Additionally, some jQuery plugins are provided (some are provided in specific cubes).

9.1.2 Development

How to change the instance logo ?

The logo is managed by css. You must provide a custom css that will contain the code below:

```
#logo {  
    background-image: url("logo.jpg");  
}
```

logo.jpg is in mycube/data directory.

How to create an anonymous user ?

This allows to browse the site without being authenticated. In the `all-in-one.conf` file of your instance, define the anonymous user as follows

```
# login of the CubicWeb user account to use for anonymous user (if you want to  
# allow anonymous)  
anonymous-user=anon  
  
# password of the CubicWeb user account matching login  
anonymous-password=anon
```

You also must ensure that this *anon* user is a registered user of the DB backend. If not, you can create through the administration interface of your instance by adding a user with in the group *guests*.

Note: While creating a new instance, you can decide to allow access to anonymous user, which will automatically execute what is described above.

How to format an entity date attribute ?

If your schema has an attribute of type *Date* or *Datetime*, you usually want to format it when displaying it. First, you should define your preferred format using the site configuration panel <http://appurl/view?vid=systempropertiesform> and then set `ui.date` and/or `ui.datetime`. Then in the view code, use:

```
entity.printable_value(date_attribute)
```

which will always return a string whatever the attribute's type (so it's recommended also for other attribute types). By default it expects to generate HTML, so it deals with rich text formatting, xml escaping...

How to update a database after a schema modification ?

It depends on what has been modified in the schema.

- update the permissions and properties of an entity or a relation: `sync_schema_props_perms('MyEntityOrRelation')`.
- add an attribute: `add_attribute('MyEntityType', 'myattr')`.
- add a relation: `add_relation_definition('SubjRelation', 'MyRelation', 'ObjRelation')`.

I get *NoSelectableObject* exceptions, how do I debug selectors ?

You just need to put the appropriate context manager around view/component selection. One standard place for components is in `cubicweb/vregistry.py`:

```
def possible_objects(self, *args, **kwargs):
    """return an iterator on possible objects in this registry for the given
    context
    """
    from logilab.common.registry import traced_selection
    with traced_selection():
        for appobjects in self.itervalues():
            try:
                yield self._select_best(appobjects, *args, **kwargs)
            except NoSelectableObject:
                continue
```

This will yield additional WARNINGS, like this:

```
2009-01-09 16:43:52 - (cubicweb.selectors) WARNING: selector one_line_rset returned 0_
↪ for <class 'cubicweb.web.views.basecomponents.WFHistoryVComponent'>
```

For views, you can put this context in `cubicweb/web/views/basecontrollers.py` in the *ViewController*:

```
def _select_view_and_rset(self, rset):
    ...
    try:
```

(continues on next page)

(continued from previous page)

```

    from logilab.common.registry import traced_selection
    with traced_selection():
        view = self._cw.vreg['views'].select(vid, req, rset=rset)
except ObjectNotFound:
    self.warning("the view %s could not be found", vid)
    req.set_message(req._("The view %s could not be found") % vid)
    vid = vid_from_rset(req, rset, self._cw.vreg.schema)
    view = self._cw.vreg['views'].select(vid, req, rset=rset)
...

```

I get “database is locked” when executing tests

If you have “database is locked” as error when you are executing security tests, it is usually because commit or rollback are missing before login() calls.

You can also use a context manager, to avoid such errors, as described here: [Managing connections or users](#).

What are hooks used for ?

Hooks are executed around (actually before or after) events. The most common events are data creation, update and deletion. They permit additional constraint checking (those not expressible at the schema level), pre and post computations depending on data movements.

As such, they are a vital part of the framework.

Other kinds of hooks, called Operations, are available for execution just before commit.

For more information, read [Hooks and Operations](#) section.

9.1.3 Configuration

How to configure a LDAP source ?

See [LDAP integration](#).

How to import LDAP users in *CubicWeb* ?

Here is a useful script which enables you to import LDAP users into your *CubicWeb* instance by running the following:

```

import os
import pwd
import sys

from logilab.database import get_connection

def getlogin():
    """avoid using os.getlogin() because of strange tty/stdin problems
    (man 3 getlogin)
    Another solution would be to use $LOGNAME, $USER or $USERNAME
    """

```

(continues on next page)

(continued from previous page)

```

"""
return pwd.getpuid(os.getuid())[0]

try:
    database = sys.argv[1]
except IndexError:
    print 'USAGE: python ldap2system.py <database>'
    sys.exit(1)

if input('update %s db ? [y/n]: ' % database).strip().lower().startswith('y'):
    cnx = get_connection(user=getlogin(), database=database)
    cursor = cnx.cursor()

    insert = ('INSERT INTO euser (creation_date, eid, modification_date, login, '
              ' firstname, surname, last_login_time, upassword) '
              "VALUES %(mtime)s, %(eid)s, %(mtime)s, %(login)s, %(firstname)s, "
              "%(surname)s, %(mtime)s, './fqEz5LeZnT6');")
    update = "UPDATE entities SET source='system' WHERE eid=%(eid)s;"
    cursor.execute("SELECT eid,type,source,extid,mtime FROM entities WHERE source!=
    ↪ 'system'")
    for eid, type, source, extid, mtime in cursor.fetchall():
        if type != 'CWUser':
            print "don't know what to do with entity type", type
            continue
        if source != 'ldapuser':
            print "don't know what to do with source type", source
            continue
        ldapinfos = dict(x.strip().split('=') for x in extid.split(','))
        login = ldapinfos['uid']
        firstname = ldapinfos['uid'][0].upper()
        surname = ldapinfos['uid'][1:].capitalize()
        if login != 'jcuissinat':
            args = dict(eid=eid, type=type, source=source, login=login,
                        firstname=firstname, surname=surname, mtime=mtime)
            print args
            cursor.execute(insert, args)
            cursor.execute(update, args)

    cnx.commit()
    cnx.close()

```

9.1.4 Security

How to reset the password for user joe ?

If you want to reset the admin password for myinstance, do:

```
$ cubicweb-ctl reset-admin-pwd myinstance
```

You need to generate a new encrypted password:

```
$ python
>>> from cubicweb.server.utils import crypt_password
>>> crypt_password('joepass')
'qH08282QN5Utg'
>>>
```

and paste it in the database:

```
$ psql mydb
mydb=> update cw_cwuser set cw_upassword='qH08282QN5Utg' where cw_login='joe';
UPDATE 1
```

if you're running over SQL Server, you need to use the CONVERT function to convert the string to varbinary(255). The SQL query is therefore:

```
update cw_cwuser set cw_upassword=CONVERT(varbinary(255), 'qH08282QN5Utg') where cw_
login='joe';
```

Be careful, the encryption algorithm is different on Windows and on Unix. You cannot therefore use a hash generated on Unix to fill in a Windows database, nor the other way round.

You can prefer use a migration script similar to this shell invocation instead:

```
$ cubicweb-ctl shell <instance>
>>> from cubicweb import Binary
>>> from cubicweb.server.utils import crypt_password
>>> crypted = crypt_password('joepass')
>>> rset = rql('Any U WHERE U is CWUser, U login "joe"')
>>> joe = rset.get_entity(0,0)
>>> joe.cw_set(upassword=Binary(crypted))
```

Please, refer to the script example is provided in the *misc/examples/chpasswd.py* file.

The more experimented people would use RQL request directly:

```
>>> rql('SET X upassword %(a)s WHERE X is CWUser, X login "joe"',
...     {'a': crypted})
```

I've just created a user in a group and it doesn't work !

You are probably getting errors such as

```
remove {'PR': 'Project', 'C': 'CWUser'} from solutions since your_user has no read_
↪access to cost
```

This is because you have to put your user in the “users” group. The user has to be in both groups.

How is security implemented ?

The basis for security is a mapping from operations to groups or arbitrary RQL expressions. These mappings are scoped to entities and relations.

This is an example for an Entity Type definition:

```
class Version(EntityType):
    """a version is defining the content of a particular project's
    release"""
    # definition of attributes is voluntarily missing
    __permissions__ = {'read': ('managers', 'users', 'guests',),
                      'update': ('managers', 'logilab', 'owners'),
                      'delete': ('managers',),
                      'add': ('managers', 'logilab',
                             ERQLEExpression('X version_of PROJ, U in_group G, '
                                              'PROJ require_permission P, '
                                              'P name "add_version", P require_group G')),
                      ↪)}

```

The above means that permission to read a Version is granted to any user that is part of one of the groups ‘managers’, ‘users’, ‘guests’. The ‘add’ permission is granted to users in group ‘managers’ or ‘logilab’ or to users in group G, if G is linked by a permission entity named “add_version” to the version’s project.

An example for a Relation Definition (RelationType both defines a relation type and implicitly one relation definition, on which the permissions actually apply):

```
class version_of(RelationType):
    """link a version to its project. A version is necessarily linked
    to one and only one project. """
    # some lines voluntarily missing
    __permissions__ = {'read': ('managers', 'users', 'guests',),
                      'delete': ('managers', ),
                      'add': ('managers', 'logilab',
                             RRQLEExpression('O require_permission P, P name "add_
↪version", '
                                              'U in_group G, P require_group G'),), }

```

The main difference lies in the basic available operations (there is no ‘update’ operation) and the usage of an RRQL-Expression (rql expression for a relation) instead of an ERQLEExpression (rql expression for an entity).

You can find additional information in the section *The security model*.

Is it possible to bypass security from the UI (web front) part ?

No. Only Hooks/Operations can do that.

Can PostgreSQL and CubicWeb authentication work with kerberos ?

If you have PostgreSQL set up to accept kerberos authentication, you can set the `db-host`, `db-name` and `db-user` parameters in the *sources* configuration file while leaving the password blank. It should be enough for your instance to connect to postgresql with a kerberos ticket.

9.2 Relation Query Language (RQL)

This chapter describes the Relation Query Language syntax and its implementation in CubicWeb.

9.2.1 Introduction

Goals of RQL

The goal is to have a semantic language in order to:

- query relations in a clear syntax
- empowers access to data repository manipulation
- making attributes/relations browsing easy

As such, attributes will be regarded as cases of special relations (in terms of usage, the user should see no syntactic difference between an attribute and a relation).

Comparison with existing languages

SQL

RQL may remind of SQL but works at a higher abstraction level (the *CubicWeb* framework generates SQL from RQL to fetch data from relation databases). RQL is focused on browsing relations. The user needs only to know about the *CubicWeb* data model he is querying, but not about the underlying SQL model.

Sparql

The query language most similar to RQL is [SPARQL](#), defined by the W3C to serve for the semantic web.

Versa

We should look in more detail, but here are already some ideas for the moment . . . *Versa* is the language most similar to what we wanted to do, but the model underlying data being RDF, there are some things such as namespaces or handling of the RDF types which does not interest us. On the functionality level, *Versa* is very comprehensive including through many functions of conversion and basic types manipulation, which we may want to look at one time or another. Finally, the syntax is a little esoteric.

Datalog

Datalog is a prolog derived query language which applies to relational databases. It is more expressive than RQL in that it accepts either *extensional* and *intensional* predicates (or relations). As of now, RQL only deals with intensional relations.

The different types of queries

Search (*Any*) Extract entities and attributes of entities.

Insert entities (*INSERT*) Insert new entities or relations in the database. It can also directly create relationships for the newly created entities.

Update entities, create relations (*SET*) Update existing entities in the database, or create relations between existing entities.

Delete entities or relationship (*DELETE*) Remove entities or relations existing in the database.

RQL relation expressions

RQL expressions apply to a live database defined by a *Yams schema*. Apart from the main type, or head, of the expression (search, insert, etc.) the most common constituent of an RQL expression is a (set of) relation expression(s).

An RQL relation expression contains three components:

- the subject, which is an entity type
- the predicate, which is a relation definition (an arc of the schema)
- the object, which is either an attribute or a relation to another entity

Warning: A relation is always expressed in the order: subject, predicate, object.

It is important to determine if the entity type is subject or object to construct a valid expression. Inverting the subject/object is an error since the relation cannot be found in the schema.

If one does not have access to the code, one can find the order by looking at the schema image in manager views (the subject is located at the beginning of the arrow).

An example of two related relation expressions:

```
P works_for C, P name N
```

RQL variables represent typed entities. The type of entities is either automatically inferred (by looking at the possible relation definitions, see [Relation definition](#)) or explicitly constrained using the `is` meta relation.

In the example above, we barely need to look at the schema. If variable names (in the RQL expression) and relation type names (in the schema) are expressively designed, the human reader can infer as much as the *CubicWeb* querier.

The `P` variable is used twice but it always represent the same set of entities. Hence `P works_for C` and `P name N` must be compatible in the sense that all the `Ps` (which *can* refer to different entity types) must accept the `works_for` and `name` relation types. This does restrict the set of possible values of `P`.

Adding another relation expression:

`P works_for C, P name N, C name "logilab"`

This further restricts the possible values of `P` through an indirect constraint on the possible values of `C`. The RQL-level [unification](#) happening there is translated to one (or several) [joins](#) at the database level.

Note: In *CubicWeb*, the term *relation* is often found without ambiguity instead of *predicate*. This predicate is also known as the *property* of the triple in [RDF concepts](#)

RQL Operators

An RQL expression's head can be completed using various operators such as `ORDERBY`, `GROUPBY`, `HAVING`, `LIMIT` etc.

RQL relation expressions can be grouped with `UNION` or `WITH`. Predicate oriented keywords such as `EXISTS`, `OR`, `NOT` are available.

The complete zoo of RQL operators is described extensively in the following chapter ([RQL syntax](#)).

9.2.2 RQL syntax

Reserved keywords

`AND, ASC, BEING, DELETE, DESC, DISTINCT, EXISTS, FALSE, GROUPBY, HAVING, ILIKE, INSERT, LIKE, LIMIT, NOT, NOW, NULL, NULLSFIRST, NULLSLAST, OFFSET, OR, ORDERBY, SET, TODAY, TRUE, UNION, WHERE, WITH`

The keywords are not case sensitive. You should not use them when defining your schema, or as RQL variable names.

Case

- Variables should be all upper-cased.
- Relation should be all lower-cased and match exactly names of relations defined in the schema.
- Entity types should start with an upper cased letter and be followed by at least a lower cased latter.

Variables and typing

Entities and values to browse and/or select are represented in the query by *variables* that must be written in capital letters.

With RQL, we do not distinguish between entities and attributes. The value of an attribute is considered as an entity of a particular type (see below), linked to one (real) entity by a relation called the name of the attribute, where the entity is the subject and the attribute the object.

The possible type(s) for each variable is derived from the schema according to the constraints expressed above and thanks to the relations between each variable.

We can restrict the possible types for a variable using the special relation **is** in the restrictions.

Virtual relations

Those relations may only be used in RQL query but are not actual attributes of your entities.

- *has_text*: relation to use to query the full text index (only for entities having fulltextindexed attributes).
- *identity*: relation to use to tell that a RQL variable is the same as another when you've to use two different variables for querying purpose. On the opposite it's also useful together with the NOT operator to tell that two variables should not identify the same entity

Literal expressions

Bases types supported by RQL are those supported by yams schema. Literal values are expressed as explained below:

- string should be between double or single quotes. If the value contains a quote, it should be preceded by a backslash '\'
- floats separator is dot '.'
- boolean values are TRUE and FALSE keywords
- date and time should be expressed as a string with ISO notation : YYYY/MM/DD [hh:mm], or using keywords TODAY and NOW

You may also use the NULL keyword, meaning 'unspecified'.

Operators

Logical operators

AND, OR, NOT, ' , '

' , ' is equivalent to 'AND' but with the smallest among the priority of logical operators (see [Operators priority](#)).

Mathematical operators

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division	4 / 2	2
%	modulo (remainder)	5 % 4	1
^	exponentiation	2.0 ^ 3.0	8
&	bitwise AND	91 & 15	11
	bitwise OR	32 3	35
#	bitwise XOR	17 # 5	20
~	bitwise NOT	~1	-2
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

Notice integer division truncates results depending on the backend behaviour. For instance, postgresql does.

Comparison operators

=, !=, <, <=, >=, >, IN

The syntax to use comparison operators is:

VARIABLE attribute <operator> *VALUE*

The = operator is the default operator and can be omitted, i.e. :

VARIABLE attribute = *VALUE*

is equivalent to

VARIABLE attribute *VALUE*

The operator *IN* provides a list of possible values:

Any X WHERE X name IN ('chauvat', 'fayolle', 'di mascio', 'thenault')

String operators

LIKE, ILIKE, ~=, REGEXP

The LIKE string operator can be used with the special character % in a string as wild-card:

-- match every entity whose name starts with 'Th'
Any X WHERE X name ~= 'Th%'
-- match every entity whose name ends with 'lt'
Any X WHERE X name LIKE '%lt'
-- match every entity whose name contains a 'l' and a 't'
Any X WHERE X name LIKE '%l%t%'

ILIKE is the case insensitive version of LIKE. It's not available on all backend (e.g. sqlite doesn't support it). If not available for your backend, ILIKE will behave like LIKE.

~= is a shortcut version of ILIKE, or of LIKE when the former is not available on the back-end.

The REGEXP is an alternative to LIKE that supports POSIX regular expressions:

```
-- match entities whose title starts with a digit
Any X WHERE X title REGEXP "^[0-9].*"
```

The underlying SQL operator used is back-end-dependent :

- the ~ operator is used for postgresql,
- the REGEXP operator for mysql and sqlite.

Other back-ends are not supported yet.

Operators priority

1. (,)
2. ^, <<, >>
3. *, /, %, &
4. +, -, |, #
5. NOT
6. AND
7. OR
8. ,

Search Query

Simplified grammar of search query:

```
[ `DISTINCT` ] `Any` V1 (, V2) \*
[ `GROUPBY` V1 (, V2) \* ] [ `ORDERBY` <orderterms> ]
[ `LIMIT` <value> ] [ `OFFSET` <value> ]
[ `WHERE` <triplet restrictions> ]
[ `WITH` V1 (, V2) \* BEING (<query>)]
[ `HAVING` <other restrictions> ]
[ `UNION` <query>]
```

Selection

The first occurring clause is the selection of terms that should be in the result set. Terms may be variable, literals, function calls, arithmetic, etc. and each term is separated by a comma.

There will be as much column in the result set as term in this clause, respecting order.

Syntax for function call is somewhat intuitive, for instance:

```
Any UPPER(N) WHERE P firstname N
```

Grouping and aggregating

The **GROUPBY** keyword is followed by a list of terms on which results should be grouped. They are usually used with aggregate functions, responsible to aggregate values for each group (see [Aggregate functions](#)).

For grouped queries, all selected variables must be either aggregated (i.e. used by an aggregate function) or grouped (i.e. listed in the **GROUPBY** clause).

Sorting

The **ORDERBY** keyword is followed by the definition of the selection order: variable or column number followed by sorting method (**ASC**, **DESC**), **ASC** being the default. If the sorting method is not specified, then the sorting is ascendant (**ASC**).

It is also possible to precise a specific ordering for *NULL* values. The **NULLSFIRST** and **NULLSLAST** options can be used to determine whether nulls appear before or after non-null values in the sort ordering. By default, null values sort as if larger than any non-null value; that is, **NULLSFIRST** is the default for **DESC** order, and **NULLSLAST** otherwise. These options are written after the sorting method when it is specified. For instance, this request will return all projects ordered by creation date in descending order, with projects with no date in last position.

```
Any X ORDERBY Y DESC NULLSLAST WHERE X creation_date Y
```

Pagination

The **LIMIT** and **OFFSET** keywords may be respectively used to limit the number of results and to tell from which result line to start (for instance, use *LIMIT 20* to get the first 20 results, then *LIMIT 20 OFFSET 20* to get the next 20).

Restrictions

The **WHERE** keyword introduces one of the “main” part of the query, where you “define” variables and add some restrictions telling what you’re interested in.

It’s a list of triplets “subject relation object”, e.g. *V1 relation (V2 | <static value>)*. Triplets are separated using [Logical operators](#).

Note: About the negation operator (**NOT**):

- **NOT X relation Y** is equivalent to **NOT EXISTS(X relation Y)**
- **Any X WHERE NOT X owned_by U** means “entities that have no relation **owned_by**”.

- Any X WHERE NOT X owned_by U, U login "syt" means “the entity have no relation owned_by with the user syt”. They may have a relation “owned_by” with another user.

In this clause, you can also use EXISTS when you want to know if some expression is true and do not need the complete set of elements that make it true. Testing for existence is much faster than fetching the complete set of results, especially when you think about using OR against several expressions. For instance if you want to retrieve versions which are in state “ready” or tagged by “priority”, you should write :

```
Any X ORDERBY PN,N
WHERE X num N, X version_of P, P name PN,
      EXISTS(X in_state S, S name "ready")
      OR EXISTS(T tags X, T name "priority")
```

not

```
Any X ORDERBY PN,N
WHERE X num N, X version_of P, P name PN,
      (X in_state S, S name "ready")
      OR (T tags X, T name "priority")
```

Both queries aren’t at all equivalent :

- the former will retrieve all versions, then check for each one which are in the matching state of or tagged by the expected tag,
- the later will retrieve all versions, state and tags (cartesian product!), compute join and then exclude each row which are in the matching state or tagged by the expected tag. This implies that you won’t get any result if the in_state or tag tables are empty (ie there is no such relation in the application). This is usually NOT what you want.

Another common case where you may want to use EXISTS is when you find yourself using DISTINCT at the beginning of your query to remove duplicate results. The typical case is when you have a multivalued relation such as Version version_of Project and you want to retrieve projects which have a version:

```
Any P WHERE V version_of P
```

will return each project number of versions times. So you may be tempted to use:

```
DISTINCT Any P WHERE V version_of P
```

This will work, but is not efficient, as it will use the SELECT DISTINCT SQL predicate, which needs to retrieve all projects, then sort them and discard duplicates, which can have a very high cost for large result sets. So the best way to write this is:

```
Any P WHERE EXISTS(V version_of P)
```

You can also use the question mark (?) to mark optional relations. This allows you to select entities related **or not** to another. It is a similar concept to [Left outer join](#):

the result of a left outer join (or simply left join) for table A and B always contains all records of the “left” table (A), even if the join-condition does not find any matching record in the “right” table (B).

You must use the ? behind a variable to specify that the relation to that variable is optional. For instance:

- Bugs of a project attached or not to a version

```
Any X, V WHERE X concerns P, P eid 42, X corrected_in V?
```

You will get a result set containing all the project's tickets, with either the version in which it's fixed or None for tickets not related to a version.

- All cards and the project they document if any

```
Any C, P WHERE C is Card, P? documented_by C
```

Notice you may also use outer join:

- on the RHS of attribute relation, e.g.

```
Any X WHERE X ref XR, Y name XR?
```

so that Y is outer joined on X by ref/name attributes comparison

- on any side of an HAVING expression, e.g.

```
Any X WHERE X creation_date XC, Y creation_date YC
HAVING YEAR(XC)=YEAR(YC)?
```

so that Y is outer joined on X by comparison of the year extracted from their creation date.

```
Any X WHERE X creation_date XC, Y creation_date YC
HAVING YEAR(XC)?=YEAR(YC)
```

would outer join X on Y instead.

Having restrictions

The HAVING clause, as in SQL, may be used to restrict a query according to value returned by an aggregate function, e.g.

```
Any X GROUPBY X WHERE X relation Y HAVING COUNT(Y) > 10
```

It may however be used for something else: In the WHERE clause, we are limited to triplet expressions, so some things may not be expressed there. Let's take an example : if you want to get people whose upper-cased first name equals to another person upper-cased first name. There is no proper way to express this using triplet, so you should use something like:

```
Any X WHERE X firstname XFN, Y firstname YFN, NOT X identity Y HAVING UPPER(XFN) =
↳UPPER(YFN)
```

Another example: imagine you want person born in 2000:

```
Any X WHERE X birthday XB HAVING YEAR(XB) = 2000
```

Notice that while we would like this to work without the HAVING clause, this can't be currently be done because it introduces an ambiguity in RQL's grammar that can't be handled by *Yapps*, the parser's generator we're using.

Sub-queries

The WITH keyword introduce sub-queries clause. Each sub-query has the form:

V1(V2) BEING (rql query)

Variables at the left of the BEING keyword defines into which variables results from the sub-query will be mapped to into the outer query. Sub-queries are separated from each other using a comma.

Let's say we want to retrieve for each project its number of versions and its number of tickets. Due to the nature of relational algebra behind the scene, this can't be achieved using a single query. You have to write something along the line of:

```
Any X, VC, TC WHERE X identity XX
WITH X, VC BEING (Any X, COUNT(V) GROUPBY X WHERE V version_of X),
      XX, TC BEING (Any X, COUNT(T) GROUPBY X WHERE T ticket_of X)
```

Notice that we can't reuse a same variable name as alias for two different sub-queries, hence the usage of 'X' and 'XX' in this example, which are then unified using the special *identity* relation (see [Virtual relations](#)).

Warning: Sub-queries define a new variable scope, so even if a variable has the same name in the outer query and in the sub-query, they technically **aren't** the same variable. So:

```
Any W, REF WITH W, REF BEING
  (Any W, REF WHERE W is Workcase, W ref REF,
    W concerned_by D, D name "Logilab")
```

could be written:

```
Any W, REF WITH W, REF BEING
  (Any W1, REF1 WHERE W1 is Workcase, W1 ref REF1,
    W1 concerned_by D, D name "Logilab")
```

Also, when a variable is coming from a sub-query, you currently can't reference its attribute or inlined relations in the outer query, you've to fetch them in the sub-query. For instance, let's say we want to sort by project name in our first example, we would have to write:

```
Any X, VC, TC ORDERBY XN WHERE X identity XX
WITH X, XN, VC BEING (Any X, COUNT(V) GROUPBY X,XN WHERE V version_of X, X name XN),
      XX, TC BEING (Any X, COUNT(T) GROUPBY X WHERE T ticket_of X)
```

instead of:

```
Any X, VC, TC ORDERBY XN WHERE X identity XX, X name XN,
WITH X, XN, VC BEING (Any X, COUNT(V) GROUPBY X WHERE V version_of X),
      XX, TC BEING (Any X, COUNT(T) GROUPBY X WHERE T ticket_of X)
```

which would result in a SQL execution error.

Union

You may get a result set containing the concatenation of several queries using the UNION. The selection of each query should have the same number of columns.

```
(Any X, XN WHERE X is Person, X surname XN) UNION (Any X,XN WHERE X is Company, X name, XN)
```

Available functions

Below is the list of aggregate and transformation functions that are supported natively by the framework. Notice that cubes may define additional functions.

Aggregate functions

COUNT(Any)	return the number of rows
MIN(Any)	return the minimum value
MAX(Any)	return the maximum value
AVG(Any)	return the average value
SUM(Any)	return the sum of values
GROUP_CONCAT(String)	return each unique value separated by a comma (for string only)

All aggregate functions above take a single argument. Take care some aggregate functions (e.g. MAX, MIN) may return *None* if there is no result row.

String transformation functions

UPPER(String)	upper case the string
LOWER(String)	lower case the string
LENGTH(String)	return the length of the string
SUBSTRING(String, start, length)	extract from the string a string starting at given index and of given length
TEXT_LIMIT_SIZE(String, max size)	If the length of the string is greater than given max size, strip it and add ellipsis ("..."). The resulting string will hence have max size + 3 characters
LIMIT_SIZE(String, format, max size)	Similar to the above, but allow to specify the MIME type of the text contained by the string. Supported formats are text/html, text/xhtml and text/xml. All others will be considered as plain text. For non plain text format, sgml tags will be first removed before limiting the string.

Date extraction functions

YEAR(Date)	return the year of a date or datetime
MONTH(Date)	return the month of a date or datetime
DAY(Date)	return the day of a date or datetime
HOURL(Datetime)	return the hours of a datetime
MINUTE(Datetime)	return the minutes of a datetime
SECOND(Datetime)	return the seconds of a datetime
WEEKDAY(Date)	return the day of week of a date or datetime. Sunday == 1, Saturday == 7.

Other functions

ABS(num)	return the absolute value of a number
RANDOM()	return a pseudo-random value from 0.0 to 1.0
FSPATH(X)	expect X to be an attribute whose value is stored in a BFSStorage and return its path on the file system
FTIRANK(X)	expect X to be an entity used in a has_text relation, and return a number corresponding to the rank order of each resulting entity
CAST(Type, X)	expect X to be an attribute and return it casted into the given final type

Examples

- Search for the object of identifier 53

```
Any X WHERE X eid 53
```

- Search material such as comics, owned by syt and available

```
Any X WHERE X is Document,
           X occurrence_of F, F class C, C name 'Comics',
           X owned_by U, U login 'syt',
           X available TRUE
```

- Looking for people working for eurocopter interested in training

```
Any P WHERE P is Person, P work_for S, S name 'Eurocopter',
           P interested_by T, T name 'training'
```

- Search note less than 10 days old written by jphc or ocy

```
Any N WHERE N is Note, N written_on D, D day> (today -10),
           N written_by P, P name 'jphc' or P name 'ocy'
```

- Looking for people interested in training or living in Paris

```
Any P WHERE P is Person, EXISTS(P interested_by T, T name 'training') OR
           (P city 'Paris')
```

- The surname and firstname of all people

```
Any N, P WHERE X is Person, X name N, X firstname P
```

Note that the selection of several entities generally force the use of “Any” because the type specification applies otherwise to all the selected variables. We could write here

```
String N, P WHERE X is Person, X name N, X first_name P
```

Note: You can not specify several types with * ... where X is FirstType or X is SecondType*. To specify several types explicitly, you have to do

```
Any X WHERE X is IN (FirstType, SecondType)
```

Insertion query

```
INSERT <entity type> V1 (, <entity type> V2) * : <assignments> [ WHERE <restriction>]
```

assignments list of relations to assign in the form *V1 relationship V2 | <static value>*

The restriction can define variables used in assignments.

Caution, if a restriction is specified, the insertion is done for *each line result returned by the restriction*.

- Insert a new person named ‘foo’

```
INSERT Person X: X name 'foo'
```

- Insert a new person named ‘foo’, another called ‘nice’ and a ‘friend’ relation between them

```
INSERT Person X, Person Y: X name 'foo', Y name 'nice', X friend Y
```

- Insert a new person named ‘foo’ and a ‘friend’ relation with an existing person called ‘nice’

```
INSERT Person X: X name 'foo', X friend Y WHERE Y name 'nice'
```

Update and relation creation queries

```
SET <assigments> [ WHERE <restriction>]
```

Caution, if a restriction is specified, the update is done for *each result line returned by the restriction*.

- Renaming of the person named ‘foo’ to ‘bar’ with the first name changed

```
SET X name 'bar', X firstname 'original' WHERE X is Person, X name 'foo'
```

- Insert a relation of type ‘know’ between objects linked by the relation of type ‘friend’

```
SET X know Y WHERE X friend Y
```


Deletion query

`DELETE (<entity type> V) | (V1 relation v2),... [WHERE <restriction>]`

Caution, if a restriction is specified, the deletion is made *for each line result returned by the restriction*.

- Deletion of the person named 'foo'

```
DELETE Person X WHERE X name 'foo'
```

- Removal of all relations of type 'friend' from the person named 'foo'

```
DELETE X friend Y WHERE X is Person, X name 'foo'
```

9.2.3 Debugging RQL

Available levels

Server debugging flags. They may be combined using binary operators.

`cubicweb.server.DBG_NONE = 0`

no debug information

`cubicweb.server.DBG_RQL = 1`

rql execution information

`cubicweb.server.DBG_SQL = 2`

executed sql

`cubicweb.server.DBG_REPO = 4`

repository events

`cubicweb.server.DBG_HOOKS = 16`

hooks

`cubicweb.server.DBG_OPS = 32`

operations

`cubicweb.server.DBG_MORE = 128`

more verbosity

`cubicweb.server.DBG_ALL = 247`

all level enabled

Enable verbose output

To debug your RQL statements, it can be useful to enable a verbose output:

```
from cubicweb import server
server.set_debug(server.DBG_RQL | server.DBG_SQL | server.DBG_ALL)
```

`cubicweb.server.set_debug(debugmode)`

change the repository debugging mode

Another example showing how to debug hooks at a specific code site:

```
from cubicweb.server import debugged, DBG_HOOKS
with debugged(DBG_HOOKS):
    person.cw_set(works_for=company)
```

Detect largest RQL queries

See *Profiling and performance* chapter (see *Profiling and performance*).

API

class cubicweb.server.debugged(*debugmode*)

Context manager and decorator to help debug the repository.

It can be used either as a context manager:

```
>>> with debugged('DBG_RQL | DBG_REPO'):
...     # some code in which you want to debug repository activity,
...     # seing information about RQL being executed an repository events.
```

or as a function decorator:

```
>>> @debugged('DBG_RQL | DBG_REPO')
... def some_function():
...     # some code in which you want to debug repository activity,
...     # seing information about RQL being executed an repository events
```

The debug mode will be reset to its original value when leaving the “with” block or the decorated function.

9.2.4 RQL usecases

Search bar

The search bar is available on a CubicWeb instance to use RQL and it’s use and configuration is described in the *doc*.

Use of RQL in Card documents - ReST

With a CubicWeb instance supporting object types with ReST content (for example [Card](#)), one can build content based on RQL queries as dynamic documents.

For this, use the *rql* and *rql-table* ReST directive, for more information about custom ReST directives [head over to the sphinx documentation](#) which uses them extensively.

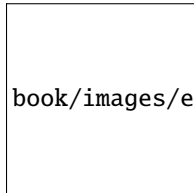
rql directive

The *rql* directive takes as input an RQL expression and a view to apply to the result.

For example, create a Card content by opening http://cubicweb_example.org/add/Card and add the following content, as an example : a table of blog entries (10 most recent blog entries table with user and date information)

```
Recent blog entries
-----

:rql:`Any B,U,D ORDERBY D DESC LIMIT 10 WHERE B is BlogEntry, B title T, B creation_date D,
↪D, B created_by U:table`
```



book/images/example-card-with-rql-directive.png

rql-table directive

rql-table enables more customization, enabling you to modify the column (*header*) contents, and the view applied for a specific column (*colvids*).

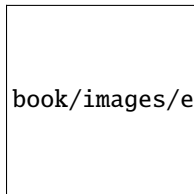
For example, create a Card content by opening http://cubicweb_example.org/add/Card and add the following content

```
Blog entries with rql-table
-----

.. rql-table::
  :vid: table
  :headers: Title with link, who wrote it, at what date
  :colvids: 1=sameetypelist

  Any B,U,D ORDERBY D DESC LIMIT 10 WHERE B is BlogEntry, B title T, B creation_date D,
  ↪B created_by U
```

All fields but the RQL string are optional. The `:headers:` option can contain empty column names.



book/images/example-card-with-rql-table-directive.png

Use in python projects and CLI

cwclientlib <<https://pypi.org/project/cwclientlib/>> enables you to use RQL in your python projects using only web requests. This project also provides a remote command line interface (CLI) you can use to replace a server side *cubicweb-ctl shell*.

Use in JavaScript/React components

cwclientelements <<https://forge.extranet.logilab.fr/open-source/cwclientelements>> is a library of reusable React components for building web application with cubicweb and RQL.

9.2.5 Implementation

BNF grammar

The terminal elements are in capital letters, non-terminal in lowercase. The value of the terminal elements (between quotes) is a Python regular expression.

```
statement ::= (select | delete | insert | update) ';'

# select specific rules
select      ::= 'DISTINCT'? E_TYPE selected_terms restriction? group? sort?

selected_terms ::= expression ( ',' expression)*

group        ::= 'GROUPBY' VARIABLE ( ',' VARIABLE)*

sort         ::= 'ORDERBY' sort_term ( ',' sort_term)*

sort_term    ::= VARIABLE sort_method =?

sort_method  ::= 'ASC' | 'DESC'

# delete specific rules
delete ::= 'DELETE' (variables_declaration | relations_declaration) restriction?

# insert specific rules
insert ::= 'INSERT' variables_declaration ( ':' relations_declaration)? restriction?

# update specific rules
update ::= 'SET' relations_declaration restriction

# common rules
variables_declaration ::= E_TYPE VARIABLE ( ',' E_TYPE VARIABLE)*

relations_declaration ::= simple_relation ( ',' simple_relation)*
```

(continues on next page)

(continued from previous page)

```

simple_relation ::= VARIABLE R_TYPE expression

restriction ::= 'WHERE' relations

relations    ::= relation (LOGIC_OP relation)*
               | '(' relations ')'

relation     ::= 'NOT'? VARIABLE R_TYPE COMP_OP? expression
               | 'NOT'? R_TYPE VARIABLE 'IN' '(' expression (',' expression)* ')'

expression   ::= var_or_func_or_const (MATH_OP var_or_func_or_const) *
               | '(' expression ')'

var_or_func_or_const ::= VARIABLE | function | constant

function     ::= FUNCTION '(' expression (',' expression) * ')'

constant     ::= KEYWORD | STRING | FLOAT | INT

# tokens
LOGIC_OP    ::= ',' | 'OR' | 'AND'
MATH_OP     ::= '+' | '-' | '/' | '*'
COMP_OP     ::= '>' | '>=' | '=' | '<=' | '<' | '~=' | 'LIKE'

FUNCTION    ::= 'MIN' | 'MAX' | 'SUM' | 'AVG' | 'COUNT' | 'UPPER' | 'LOWER'

VARIABLE    ::= '[A-Z][A-Z0-9]*'
E_TYPE     ::= '[A-Z]\w*'
R_TYPE     ::= '[a-z_]+'

KEYWORD     ::= 'TRUE' | 'FALSE' | 'NULL' | 'TODAY' | 'NOW'
STRING      ::= '"'([^\]|\\\.)*'"' | "'"([^\"]|\\\.)*\'"'
FLOAT       ::= '\d+\.\d*'
INT         ::= '\d+'

```

Internal representation (syntactic tree)

The tree research does not contain the selected variables (e.g. there is only what follows “WHERE”).

The insertion tree does not contain the variables inserted or relations defined on these variables (e.g. there is only what follows “WHERE”).

The removal tree does not contain the deleted variables and relations (e.g. there is only what follows the “WHERE”).

The update tree does not contain the variables and relations updated (e.g. there is only what follows the “WHERE”).

```

Select      ((Relationship | And | Or)?, Group?, Sort?)
Insert      (Relations | And | Or)?
Delete      (Relationship | And | Or)?
Update      (Relations | And | Or)?

And         ((Relationship | And | Or), (Relationship | And | Or))

```

(continues on next page)

(continued from previous page)

```

Or                ((Relationship | And | Or), (Relationship | And | Or))

Relationship      ((VariableRef, Comparison))

Comparison        ((Function | MathExpression | Keyword | Constant | VariableRef) +)

Function          (())

MathExpression    ((MathExpression | Keyword | Constant | VariableRef), (MathExpression |
↳Keyword | Constant | VariableRef))

Group             (VariableRef +)

Sort              (SortTerm +)

SortTerm          (VariableRef +)

VariableRef       ()

Variable          ()

Keyword           ()

Constant          ()

```

Known limitations

- The current implementation does not support linking two relations of type ‘is’ with an OR. I do not think that the negation is supported on this type of relation (XXX to be confirmed).
- missing COALESCE and certainly other things...
- writing an rql query requires knowledge of the used schema (with real relation names and entities, not those viewed in the user interface). On the other hand, we cannot really bypass that, and it is the job of a user interface to hide the RQL.

Topics

It would be convenient to express the schema matching relations (non-recursive rules):

```

Document class Type <-> Document occurrence_of Fiche class Type
Sheet class Type <-> Form collection Collection class Type

```

Therefore 1. becomes:

```

Document X where
X class C, C name 'Cartoon'
X owned_by U, U login 'syt'
X available true

```

I’m not sure that we should handle this at RQL level ...

There should also be a special relation ‘anonymous’.

9.3 Introducing Mercurial

9.3.1 Introduction

Mercurial manages a distributed repository containing revisions trees (each revision indicates the changes required to obtain the next, and so on). Locally, we have a repository containing revisions tree, and a working directory. It is possible to put in its working directory, one of the versions of its local repository, modify and then push it in its repository. It is also possible to get revisions from another repository or to export its own revisions from the local repository to another repository.

In contrast to CVS/Subversion, we usually create a repository per project to manage.

In a collaborative development, we usually create a central repository accessible to all developers of the project. These central repository is used as a reference. According to their needs, everyone can have a local repository, that they will have to synchronize with the central repository from time to time.

9.3.2 Major commands

- Create a local repository:

```
hg clone ssh://myhost//home/src/repo
```

- See the contents of the local repository (graphical tool in Qt):

```
hgview
```

- Add a sub-directory or file in the current directory:

```
hg add subdir
```

- Move to the working directory a specific revision (or last revision) from the local repository:

```
hg update [identifier-revision]
hg up [identifier-revision]
```

- Get in its local repository, the tree of revisions contained in a remote repository (this does not change the local directory):

```
hg pull ssh://myhost//home/src/repo
hg pull -u ssh://myhost//home/src/repo # equivalent to pull + update
```

- See what are the heads of branches of the local repository if a *pull* returned a new branch:

```
hg heads
```

- Submit the working directory in the local repository (and create a new revision):

```
hg commit
hg ci
```

- Merge with the mother revision of local directory, another revision from the local repository (the new revision will be then two mothers revisions):

```
hg merge identifier-revision
```

- Export to a remote repository, the tree of revisions in its content local repository (this does not change the local directory):

```
hg push ssh://myhost//home/src/repo
```

- See what local revisions are not in another repository:

```
hg outgoing ssh://myhost//home/src/repo
```

- See what are the revisions of a repository not found locally:

```
hg incoming ssh://myhost//home/src/repo
```

- See what is the revision of the local repository which has been taken out from the working directory and amended:

```
hg parent
```

- See the differences between the working directory and the mother revision of the local repository, possibly to submit them in the local repository:

```
hg diff
hg commit-tool
hg ct
```

9.3.3 Best Practices

- **Remember to *hg pull -u* regularly, and particularly before a *hg commit*.**
- Remember to *hg push* when your repository contains a version relatively stable of your changes.
- If a *hg pull -u* created a new branch head:
 1. find its identifier with *hg head*
 2. merge with *hg merge*
 3. *hg ci*
 4. *hg push*

9.3.4 More information

For more information about Mercurial, please refer to the Mercurial project online [documentation](#).

9.4 Installation dependencies

When you run CubicWeb from source, either by downloading the tarball or cloning the mercurial tree, here is the list of tools and libraries you need to have installed in order for CubicWeb to work:

- yapps - <http://theory.stanford.edu/~amitp/yapps/> - <http://pypi.python.org/pypi/Yapps2>
- pygraphviz - <http://networkx.lanl.gov/pygraphviz/> - <http://pypi.python.org/pypi/pygraphviz>
- docutils - <http://docutils.sourceforge.net/> - <http://pypi.python.org/pypi/docutils>
- lxml - <http://codespeak.net/lxml> - <http://pypi.python.org/pypi/lxml>

- logilab-common - <https://www.logilab.org/project/logilab-common> - <http://pypi.python.org/pypi/logilab-common/>
- logilab-database - <https://www.logilab.org/project/logilab-database> - <http://pypi.python.org/pypi/logilab-database/>
- logilab-constraint - <https://www.logilab.org/project/logilab-constraint> - <http://pypi.python.org/pypi/constraint/>
- logilab-mtconverter - <https://www.logilab.org/project/logilab-mtconverter> - <http://pypi.python.org/pypi/logilab-mtconverter/>
- rql - <https://www.logilab.org/project/rql> - <http://pypi.python.org/pypi/rql>
- yams - <https://www.logilab.org/project/yams> - <http://pypi.python.org/pypi/yams>
- indexer - <https://www.logilab.org/project/indexer> - <http://pypi.python.org/pypi/indexer>
- passlib - <https://code.google.com/p/passlib/> - <http://pypi.python.org/pypi/passlib>

If you're using a Postgresql database (recommended):

- psycopg2 - <http://initd.org/projects/psycopg2> - <http://pypi.python.org/pypi/psycopg2>

Other optional packages:

- fyzz - <https://www.logilab.org/project/fyzz> - <http://pypi.python.org/pypi/fyzz> to activate *Sparql querying*

Any help with the packaging of CubicWeb for more than Debian/Ubuntu (including eggs, buildouts, etc) will be greatly appreciated.

9.5 Javascript docstrings

Whereas in Python source code we only need to include a module docstrings using the directive `.. automodule:: mypythonmodule`, we will have to explicitly define Javascript modules and functions in the docstrings since there is no native directive to include Javascript files.

9.5.1 Rest generation

pyjsrest is a small utility parsing Javascript docstrings and generating the corresponding Restructured file used by Sphinx to generate HTML documentation. This script will have the following structure:

```
=====
filename.js
=====
.. module:: filename.js
```

We use the `.. module::` directive to register a javascript library as a Python module for Sphinx. This provides an entry in the module index.

The contents of the docstring found in the javascript file will be added as is following the module declaration. No treatment will be done on the doctring. All the documentation structure will be in the docstrings and will comply with the following rules.

9.5.2 Docstring structure

Basically we document javascript with RestructuredText docstring following the same convention as documenting Python code.

The doctring in Javascript files must be contained in standard Javascript comment signs, starting with `/**` and ending with `*/`, such as:

```
/**
 * My comment starts here.
 * This is the second line prefixed with a `*`.
 * ...
 * ...
 * All the follwing line will be prefixed with a `*` followed by a space.
 * ...
 * ...
 */
```

Comments line prefixed by `//` will be ignored. They are reserved for source code comments dedicated to developers.

9.5.3 Javascript functions docstring

By default, the *function* directive describes a module-level function.

function directive

Its purpose is to define the function prototype such as:

```
.. function:: loadxhtml(url, data, reqtype, mode)
```

If any namespace is used, we should add it in the prototype for now, until we define an appropriate directive:

```
.. function:: jquery.fn.loadxhtml(url, data, reqtype, mode)
```

Function parameters

We will define function parameters as a bulleted list, where the parameter name will be backquoted and followed by its description.

Example of a javascript function docstring:

```
.. function:: loadxhtml(url, data, reqtype, mode)

cubicweb loadxhtml plugin to make jquery handle xhtml response

fetches `url` and replaces this's content with the result

Its arguments are:

* `url`

* `mode`, how the replacement should be done (default is 'replace')
```

(continues on next page)

(continued from previous page)

Possible values are :

- 'replace' to replace the node's content with the generated HTML
- 'swap' to replace the node itself with the generated HTML
- 'append' to append the generated HTML to the node's content

Optional parameter specification

Javascript functions handle arguments not listed in the function signature. In the javascript code, they will be flagged using `/*...*/`. In the docstring, we flag those optional arguments the same way we would define it in Python:

```
.. function:: asyncRemoteExec(fname, arg1=None, arg2=None)
```


CHANGELOG

10.1 3.36.15 (2023-03-24)

10.1.1 Bug fixes

- testlib: define properly a generate_tzdatetime method with timezone (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/716>)

10.1.2 Continuous integration

- avoid launching duplicated migrations tests
- clean CI of unused jobs
- disable can-i-merge
- don't wait for tests to start QA jobs
- smoke_test: add timeout to request to avoid hanging up for too long
- smoke_test: handle ConnectionError situation
- test-cube-skeleton: ensure we use the same python version for smoke test than py3-* tests

10.2 3.36.14 (2023-03-02)

10.2.1 Bug fixes

- sphinx-theme 1.0 breaks doc build

10.3 3.36.13 (2023-03-02)

10.3.1 Bug fixes

- make sure we only install yapps2-logilab by updating dependencies
- tried to format a string while missing one formatting argument

10.4 3.36.12 (2023-01-17)

10.4.1 New features

- skeleton: remove format=pylint option from tox because it's better without it

10.5 3.36.11 (2023-01-12)

10.5.1 Bug fixes

- avoid risking new cubes to install pre-release version of black
- formrenderers: use UStringIO instead of list to keep the same api as self.w (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/597>)
- schema_exporters: Add missing description field for relations (e.g *in_state*) to schema exporter

10.6 3.36.10 (2022-11-15)

10.6.1 Bug fixes

- hook: correct a typo, self.warn doesn't exist

10.7 3.36.9 (2022-10-04)

10.7.1 Bug fixes

- attr: when an entity is not existing always return None when fetching its attributes (#599)
- web.views: escape text from the undohistory view (#598)

10.7.2 Various changes

- delete unused translations from *.po files (#600)
- skeleton: add *long_description_content_type* in setup.py

10.8 3.36.8 (2022-09-22)

10.8.1 Bug fixes

- startup_views: raise AuthenticationError if anon access is disabled on StartupView (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/595>)

10.9 3.36.7 (2022-09-14)

10.9.1 Bug fixes

- bookmark: do not escape the xaddrelation view from ajaxedit module

10.10 3.36.6 (2022-08-30)

10.10.1 Bug fixes

- pyramid: Redirect to the wanted URL after a successfully loggedin (to #584)
- xss: Ensure to use the xml_escape method on entity attributes
- perf: Restore initial performances by removing the unnecessary join

10.11 3.36.5 (2022-07-21)

10.11.1 Bug fixes

- schema_exporters: handle symmetrical relation in schema export. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/568>)

10.11.2 Various changes

- feat(markdown)!: update Markdown version to 3.4 and rewrite urlize extension (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/569>)

10.12 3.36.4 (2022-07-13)

- merge 3.35.6 into 3.36

10.13 3.36.3 (2022-06-03)

10.13.1 Bug fixes

- pyramid: adapt TestApp.put_json method to CSRF

10.14 3.36.2 (2022-06-01)

10.14.1 New features

- pkg: upgrade version of waitress to 2.1.1 or more, for security reason. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/543>)

10.14.2 Bug fixes

- base64.decodestring is deprecated and has been removed
- pyramid: adapt TestApp.post_json method to CSRF
- pyramid: try to get “/login” if “/” is forbidden

10.15 3.36.1 (2022-03-31)

10.15.1 Bug fixes

- rql2sql: upgrade RQL version to fix translation of NOT EXISTS(X eid Y) (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/528>)
- view: don't escape html tags inside image previews

10.16 3.36.0 (2022-03-14)

10.16.1 New features

- markdown: load extra extensions to render tables (#515)
- schema_exporter: add a parameter to export schema as dict (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/522>)

10.16.2 Documentation

- fix sidebar table of content
- improve basic tutorial
- improve home and sidebar
- improve setup instructions
- improve skeleton readme
- set version number
- use relative links for static resources
- use right number of characters for titles
- use sphinx_book_theme

10.17 3.35.12 (2022-11-15)

10.17.1 Bug fixes

- hook: correct a typo, self.warn doesn't exist

10.18 3.35.11 (2022-10-04)

10.18.1 Bug fixes

- attr: when an entity is not existing always return None when fetching its attributes (#599)
- web.views: escape text from the undohistory view (#598)

10.18.2 Various changes

- delete unused translations from *.po files (#600)
- skeleton: add *long_description_content_type* in setup.py

10.19 3.35.10 (2022-09-22)

10.19.1 Bug fixes

- startup_views: raise AuthenticationError if anon access is disabled on StartupView (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/595>)

10.20 3.35.9 (2022-09-14)

10.20.1 Bug fixes

- bookmark: do not escape the xaddrelation view from ajaxedit module

10.21 3.35.8 (2022-08-30)

10.21.1 Bug fixes

- pyramid: Redirect to the wanted URL after a successfully loggedin (to #584)
- xss: Ensure to use the xml_escape method on entity attributes
- perf: Restore initial performances by removing the unnecessary join

10.22 3.35.7 (2022-07-21)

10.22.1 Bug fixes

- `schema_exporters`: handle symmetrical relation in schema export. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/568>)

10.22.2 Various changes

- `feat(markdown)!`: update Markdown version to 3.4 and rewrite urlize extension (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/569>)

10.23 3.35.6 (2022-07-13)

10.23.1 Various changes

- fix warnings of yams 0.48+ (3.35 requires <0.48)

10.24 3.35.5 (2022-07-13)

10.24.1 Bug fixes

- `basecontrollers`: str object have no more “decode” method since py3
- fix some warnings of yams 0.48+
- `htmlwidgets`: `BoxLink` rendering is broken
- `server`: remove a memory leak related to a file
- `test`: improve one related to CSRF

10.25 3.35.4 (2022-06-03)

10.25.1 Bug fixes

- `pyramid`: adapt `TestApp.put_json` method to CSRF

10.26 3.35.3 (2022-06-01)

10.26.1 New features

- pkg: upgrade version of waitress to 2.1.1 or more, for security reason. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/543>)

10.26.2 Bug fixes

- base64.decodestring is deprecated and has been removed
- pyramid: adapt TestApp.post_json method to CSRF
- pyramid: try to get “/login” if “/” is forbidden

10.27 3.35.2 (2022-03-31)

10.27.1 Bug fixes

- rql2sql: upgrade RQL version to fix translation of NOT EXISTS(X eid Y) (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/528>)
- view: don't escape html tags inside image previews

10.28 3.35.1 (2022-03-09)

- avoid escaping cubicweb:loadurl's value twice (to #523.)
- don't escape whole key="value" attributes in TreeViewItemView (to #523.)
- escape URLs passed as href attributes (to #523.)

10.29 3.35 (2022-02-02)

10.29.1 Breaking changes

- deprecate RQLSuggestionsBuilder component ; users of this component should now use *rqlsuggestions.RQLSuggestionsBuilder* instead. RQL bar completion behaviour can be changed by replacing the “rql_suggest” ajax function. If this function isn't registered, rql completion is disabled.
- remove RQLNoSuggestionsBuilder
- disable login using GET requests for security reasons
- web: remove support of old Internet Explorer versions: *add_css* no longer accepts *iespec* and *ieonly* arguments

10.29.2 New features

- add a Dockerfile in the skeleton
- add a function for deleting entities faster
- config: add help messages in configuration files (all-in-one and sources)
- disable constraints checks on the DB upon deletion
- show cube name when there is a version conflict
- skeleton: add release-new in skeleton
- upgrade to yams 0.47
- content negotiation: we now can use `/<etype>/<rest_attr>` route for content negotiation, if `rest_attr` is defined, the route `/<etype>/<rest_attr>` is disabled for content negotiation in this situation

10.29.3 Bug fixes

- `relation_type` not existing in some conditions on `RelationDefinition` (`ionDefinition.rtype` has been deprecated in yams in favor of `relation_type`)

10.29.4 Continuous integration

- only collect warnings when running tests on the default branch (#489.)

10.30 3.34.3 (2022-03-31)

10.30.1 Bug fixes

- rql2sql: upgrade RQL version to fix translation of NOT EXISTS(X eid Y) (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/528>)
- view: don't escape html tags inside image previews

10.31 3.34.2 (2022-03-09)

10.31.1 Bug fixes

- avoid escaping `cubicweb:loadurl`'s value twice (to #523.)
- don't escape whole `key="value"` attributes in `TreeViewItemView` (to #523.)
- escape URLs passed as href attributes (to #523.)

10.32 3.34.1 (2021-12-01)

10.32.1 Bug fixes

- server: correct RQL generation when we have function in ORDERBY (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/466>)

10.33 3.34.0 (2021-11-23)

10.33.1 Breaking changes

- Python 3.7 is now the minimum supported version of Python;
- test: `settings = {"cubicweb.bwcompat": true}` is now the default for test, please, check your test if they are failing because of this;
- test: the qunit test driver has been removed.
- remove our deprecated and unused wsgi module
- fix!(handler): redirection to login on `cubicweb.AuthenticationError` Previously we were sending a forbidden response (403) with the login form as the html content, now we redirect (303) to the login form instead.

Since we updated `rdflib` to version 6, some packages like `rdflib_jsonld` are no longer needed. Please, check your dependencies if you have any issue.

10.33.2 New features

- add an `export-schema` command to `cwctl`
- csrf: add debug login when creating a new csrf token
- `get_cleaned_form_data`: add backward deprecated compatibility on `req.form`
- `pyramid/test/ux`: better debugging information when failing to get CSRF token
- security: implement enforcing form validation for POST arguments
- store: allow stores to be used as context manager (#446)

10.33.3 Bug fixes

- allow in `Int` as `rest_attr`
- `build_doc`: `docutils` version 0.18.0 breaks doc building
- `doc8`: indentation was using tabs in `3.32_reledit.rst`
- `ldap`: upgrade to `ldap3` (datetime, encode fix)
- `p3-misc`: missing `fyzz` modules for certain tests in `spa2rql`
- `pyramid/test`: `webapp` handles cookies for us, we don't need to manually set them
- `RDFLib`: Remove `rdflib-jsonld` dependency and use `RDFLib v6 jsonld` builtin parser **BREAKING CHANGE**: The `RDFLib v6` does not support python 3.6 anymore. With this dependency, CubicWeb neither.
- `reledit`: Do not retrieve a list of schemata with `_compute_ttypes`

- remove qunit test stuff (#447.)
- skeleton: use forge.extranet.logilab.fr as default web url for new cubes (#463)
- startup: Fix RQL query to take advantage of caching (#384)
- store csrf token during login
- test: make anonymous user creation hook tests pass (#452.)
- utils: remove an useless space character
- views: remove unneeded xml_escape for primary titles

10.33.4 Continuous integration

- allow sonarqube to fails until we fix the internal url problem
- test: don't wait lint to run tests (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/445>)
- use some gitlab-ci-templates (#455)
- uses buster-slim-pgl11-firefox custom image for py3-auto-test job
- uses cubicweb/dockerfiles/can-i-merge image to optimize can-i-merge job

10.33.5 Various changes

- instance-config: add attributes for authenticated smtp
- pyramid.compat is deprecated and will be removed in Pyramid 2.0. The functionality is no longer necessary, as Pyramid 2.0 drops support for Python 2.
- webconfig: remove an unused configuration option

10.34 3.33.13 (2022-03-09)

10.34.1 Bug fixes

- avoid escaping cubicweb:loadurl's value twice (to #523.)
- don't escape whole key="value" attributes in TreeViewItemView (to #523.)
- escape URLs passed as href attributes (to #523.)

10.35 3.33.12 (2021-12-01)

10.35.1 Bug fixes

- server: correct RQL generation when we have function in ORDERBY (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/466>)

10.36 3.33.11 (2021-11-17)

10.36.1 Bug fixes

- pkg: pin Yams version < 0.46.0

10.37 3.33.10 (2021-11-17)

- Removed allowed-http-host-headers configuration (which was a breaking change), since we don't have this vulnerability in CubicWeb.

10.37.1 Various changes

- depend on sphinx>=4.3

10.38 3.33.9 (2021-11-08)

10.38.1 Bug fixes

- views: remove an abusive escape (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/457>)

10.39 3.33.8 (2021-11-02)

10.39.1 Bug fixes

- build_doc: docutils version 0.18.0 breaks doc building (#443)
- pkg: don't use pyparsing 3 since it's not compatible with rdflib 5. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/441>)
- pkg: pin pyramid_multiauth version to avoid compatibility issue with Pyramid 1 (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/450>)

10.40 3.33.7 (2021-10-12)

10.40.1 Bug fixes

- ldap: upgrade to ldap3 (datetime, encode fix)
- csrf: ensure that we have a csrf token returned on every requests

10.41 3.33.6 (2021-10-04)

10.41.1 Bug fixes

- facet: remove abusive escaping in facets views. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/394>)

10.42 3.33.5 (2021-09-29)

10.42.1 Bug fixes

- backout “limit setupools version to avoid issue with 2to3”
- use our package rdflib-jsonld-without-2to3, this is a fork of rdflib-jsonld with 2to3 usage removed, but which still contains the whole package code unlike rdflib-jsonld 0.6.x.
- reledit: Do not retrieve a list of schemata with `_compute_ttypes`

10.43 3.33.4 (2021-09-24)

- don’t escape value in navigation components (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/389>)
- views: remove unneeded `xml_escape` for primary titles
- setup: keep rdflib-jsonld at version < 0.6.0
- setup: limit setupools version to avoid issue with 2to3

10.44 3.33.3 (2021-09-14)

- upgrade rdflib-jsonld version to keep compatibility with setupools 58 and above

10.44.1 Bug fixes

- startup: Fix RQL query to take advantage of caching (#384)

10.45 3.33.2 (2021-09-02)

10.45.1 Documentation

- tuto: Fix path

10.46 3.33.1 (2021-08-31)

10.46.1 New features

- allowed-http-host-headers: automatically add default hostname to the allowed list on debug mode
- req: Add a “limit” parameter to RequestSessionBase.find
- req: Add exists for optimized search of at least one entity
- ux: better error message when a controller can’t be select

10.46.2 Bug fixes

- fyzz dep was missing for running certain tests
- only fyzz 0.2.2 is compatible with python 3
- typo: fix some misspellings

10.46.3 Documentation

- fix allowed-http-host-header label and quote from Django’s doc

10.46.4 Continuous integration

- integrate can-i-merge

10.46.5 Various changes

- 3.33: improve changelog quality
- fix(bwcompat)!: return a 400 instead of a 401 when failed to select a controller
- fix: allowed-http-host-headers has been released in 3.33 actually
- misc: fix rst syntax

10.47 3.33.0 (2021-08-03)

10.47.1 New features

- **BREAKING** security: introduce allowed-http-host-header against host attack (However, this is backed out in 3.33.10).
- add postgresql extra requires
- config: add ‘debug’ option in “[main]” of all-in-one.conf that does the same thing than “-D” in “cubicweb-ctl pyramid”
- rich: use rich.traceback to have nicer tracebacks <https://github.com/willmcgugan/rich/>

10.47.2 Bug fixes

- add default value for params argument in pyramid webtest post function (#350)
- csrf: give CSRF token when using /ajax route
- empty identification cookie on webapp.reset()
- pin rdflib < 6.0.0 to avoid compatibility issues
- rdf: graph.serialize needs to encode its content in utf-8
- security: change configuration [WEB]interface default value to 127.0.0.1
- views: Fix reledit errors when trying modify relation with multi subjects

10.47.3 Continuous integration

- use image from heptapod registry since r.intra was shut down

10.47.4 Various changes

- use open-source/gitlab-ci-templates in cube skeleton
- drop mention of MySQL and SQLServer support
- update cube installation procedure documentation
- remove *.spec from skeleton

10.48 3.32.14 (2021-12-01)

10.48.1 Bug fixes

- server: correct RQL generation when we have function in ORDERBY (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/466>)

10.49 3.32.13 (2021-11-17)

10.49.1 Bug fixes

- pkg: pin Yams version < 0.46.0

10.50 3.32.12 (2021-11-17)

10.50.1 Various changes

- depend on sphinx>=4.3

10.51 3.32.11 (2021-11-08)

10.51.1 Bug fixes

- views: remove an abusive escape (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/457>)

10.52 3.32.10 (2021-11-02)

10.52.1 Bug fixes

- build_doc: docutils version 0.18.0 breaks doc building (#443)
- pkg: don't use pyparsing 3 since it's not compatible with rdflib 5. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/441>)
- pkg: pin pyramid_multiauth version to avoid compatibility issue with Pyramid 1 (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/450>)

10.53 3.32.9 (2021-10-12)

10.53.1 Bug fixes

- csrf: ensure that we have a csrf token returned on every requests

10.54 3.32.8 (2021-10-04)

10.54.1 Bug fixes

- facet: remove abusive escaping in facets views. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/394>)

10.55 3.32.7 (2021-09-29)

10.55.1 Bug fixes

- backout “limit setuptools version to avoid issue with 2to3”
- use our package rdflib-jsonld-without-2to3, this is a fork of rdflib-jsonld with 2to3 usage removed, but which still contains the whole package code unlike rdflib-jsonld 0.6.x.
- reledit: Do not retrieve a list of schemata with `_compute_ttypes`

10.56 3.32.6 (2021-09-24)

10.56.1 Bug fixes

- don’t escape value in navigation components (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/389>)
- views: remove unneeded `xml_escape` for primary titles
- setup: keep rdflib-jsonld at version < 0.6.0
- setup: limit setuptools version to avoid issue with 2to3

10.57 3.32.5 (2021-09-14)

- upgrade rdflib-jsonld version to keep compatibility with setupools 58 and above

10.58 3.32.4 (2021-09-02)

10.58.1 Bug fixes

- do not use `localhost.local` as test domain, but keep the one already defined

10.59 3.32.3 (2021-08-31)

10.59.1 New features

- migration: add a migration script to warn about incompatibility of `cwtags`. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/367>)

10.59.2 Bug fixes

- bringing back CubicWebServerTC and porting it to pyramid
- fix bad escaped values in web views
- pkg: since we added csrf mechanism, we need pyramid ≥ 1.9
- test_newcube were broken because we removed cubicweb-*.spec file but didn't updated the tests

10.60 3.32.2 (2021-07-30)

10.60.1 New features

- use open-source/gitlab-ci-templates in cube skeleton

10.60.2 Bug fixes

- add default value for params argument of PyramidCWTest.webapp.post (#350)
- csrf: give CSRF token when using /ajax route
- empty identification cookie on webapp.reset()
- remove *.spec from skeleton
- views: Fix reledit errors when trying modify relation with multi subjects

10.61 3.32.1 (2021-07-23)

10.61.1 Bug fixes

- pin rdflib $< 6.0.0$ to avoid compatibility issues

10.61.2 Continuous integration

- use image from heptapod registry since r.intra was shut down

10.62 3.32.0 (2021-07-13)

10.62.1 Security, breaking changes

Protection against XSS

self.w API has been changed to automatically escape arguments used to format the string to mitigate XSS attacks.

This means that instead of writing:

```
self.w("some %s string %s" % (a, b))
```

You need to write:

```
self.w("some %s string %s", a, b)
```

And CubicWeb will escape all arguments given to `self.w` which are `a` and `b` here.

If for a specific reason (for example generating javascript) you don't want to escape the arguments of `self.w` you can use the `escape` kwarg argument of `self.w` like this:

```
self.w("some %s string %s", a, b, escape=False)
```

This is normally retrocompatible since `self.w` old API with only one argument still works (but you **shouldn't** use it anymore) but if you have been giving a custom function as `self.w` you'll need to adapt the API of this function to match `self.w` new API which is:

```
def w(self, string, *args, **kwargs, escape=False): ...
```

Also note that `UStringIO.write` function has also been modified to be compatible with `self.w` new API (so if you are using it you won't need to port this code).

CSRF protection

A CSRF protection mechanism has been integrated in CubicWeb using Pyramid CSRF built-in protection. Regarding breaking changes:

- Cubicweb now **only works with pyramid**
- if you are only using cubicweb “web” without ajax and you have been doing advanced modification at the session management level this shouldn't break anything for you
- if you are doing POST/PUT/DELETE... requests using AJAX, you need to adapt your code to send the `csrf_token` otherwise all you requests will be denied. This is explained in the AJAX section of the documentation: `csrf_protection`

The whole mechanism is explained in the documentation: `csrf_protection`

10.62.2 Other breaking changes

We decided to stop releasing cubicweb as debian packages that we used on multi-purpose servers in favor of docker images that we run with docker-compose or on kubernetes. Thanks for all the fishes.

10.62.3 New features

- add a component to disable RQL suggestions: `cubicweb.web.views.magicsearch.RQLNoSuggestionsBuilder`

10.62.4 Bug fixes

- [reledit] display reledit for a relation if some conditions are satisfied ([1] the relation don't have rqlexpr permissions and can be deleted [2] at least one of related entites can be deleted)
- pyramid/predicates: avoid to show an error without a session connection
- be sure db-statement-timeout is not None
- correctly transform cubicweb.web.RemoteCallFailed into pyramid corresponding exceptions, this allow to propagate the correct content type (for example for json exceptions)
- "cubicweb-ctl list" now supports multiple dependencies constraints

10.62.5 Various changes

- fix error cases when internationalizable is not defined on rdef
- improve docstring in web.views.basecontrollers

10.62.6 Continuous integration

- coverage: gitlab-ci is able to read the coverage report we produce
- disable from-forge for now since we aren't using them
- fix path to coverage-*.xml for non-reports artifacts
- flake8: integrate flake8-gl-codeclimate for QA reports
- integrate junit reports style for tests errors in gitlab
- optimisation: allow to interrupt started jobs that can be replaced
- pytest-html: generate self contained html file for easier test repport browsing
- trigger py3-* jobs on tox.ini/.gitlab-ci.yml/requirements modifications
- use gitlab readthedocs integration

10.62.7 Developer experience

- using black on the whole project o/ (thx for hg format-source)
- debug/ux: display traceback of stderr when exception in addition of the html page
- testing: activate debug mode during testing
- ux: display on stdout the requests made to the server like nginx
- ux: display traceback on stderr on request failure
- ux: logger.info for selected view by ViewController

10.63 3.31.9 (2021-11-17)

10.63.1 Bug fixes

- pkg: pin Yams version < 0.46.0

10.64 3.31.8 (2021-11-17)

10.64.1 Various changes

- depend on sphinx>=4.3

10.65 3.31.7 (2021-11-02)

10.65.1 Bug fixes

- build_doc: docutils version 0.18.0 breaks doc building (#443)
- pkg: don't use pyparsing 3 since it's not compatible with rdflib 5. (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/441>)
- pkg: pin pyramid_multiauth version to avoid compatibility issue with Pyramid 1 (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/450>)

10.66 3.31.6 (2021-09-28)

10.66.1 Various changes

- backout “limit setuptools version to avoid issue with 2to3”
- use our package rdflib-jsonld-without-2to3, this is a fork of rdflib-jsonld with 2to3 usage removed, but which still contains the whole package code unlike rdflib-jsonld 0.6.x.

10.67 3.31.5 (2021-09-24)

10.67.1 Bug fixes

- setup: keep rdflib-jsonld at version < 0.6.0
- setup: limit setuptools version to avoid issue with 2to3

10.68 3.31.4 (2021-09-14)

- upgrade rdflib-jsonld version to keep compatibility with setupools 58 and above

10.69 3.31.3 (2021-07-23)

10.69.1 Bug fixes

- pin rdflib < 6.0.0 to avoid compatibility issues

10.69.2 Continuous integration

- use image from heptapod registry since r.intra was shut down

10.70 3.31.2 (2021-07-19)

10.70.1 Bug fixes

- do not consume a lot of time to collect debug data if no one is listening on debug channels
- fix incomplete merge of previous versions (brings back *write_front*)

10.71 3.31.1 (2021-05-18)

10.71.1 Revert

- backed out changeset bcb633bd791d, don't give event to notify_on_commit Notification are done using Operation, and Operation do not have a event attribute, because they can be used for several event. Moreover, this commit (bcb633bd791d) changed the prototype of the notify_on_commit without giving the right event to the Operation (which is a singleton).

10.72 3.31 (2021-05-04)

10.72.1 New features

- handle same_site cookies configuration in pyramid.ini
- order: add support for order by NULLS LAST and NULLS FIRST
- improve default cubicweb skeleton

10.72.2 Bug fixes

- create anonymous user at runtime if it doesn't exist already.
- dbcreate: don't ask confirmation to create schema in automatic
- hooks/notification: BREAKING CHANGE correctly initialize operation with event attribute
- RQLEExpression: performance issue on RQLEExpressions using EXISTS() *BREAKING CHANGE*: explicitly use EXISTS in RQLEExpression for permissions
- fix some security issues

10.72.3 Documentation

- tuto: add structure of “enhance views” museum tutorial part.
- tuto: redact “React in a CubicWeb view” museum tuto part.
- tuto: rename cubicweb-tuto to tuto, avoiding confusion with cubicweb_tuto

10.72.4 Continuous integration

- gitlab-ci: set expiration delay to 2 weeks for artifacts
- image is no longer a global keyword, use default
- rename jobs names to match global conventions
- test skeleton own tox in the CI

10.72.5 Various changes

- cleanup: Remove migrations for 3.21 and less
- py3: Rename *raw_input* (that does not exist anymore) to *input*
- tests: create a .nobackup file in the data/database directory (<https://forge.extranet.logilab.fr/cubicweb/cubicweb/-/issues/298>)

10.73 3.30.1 (2021-07-23)

10.73.1 Bug fixes

- pin rdflib < 6.0.0 to avoid compatibility issues

10.73.2 Continuous integration

- use image from heptapod registry since r.intra was shut down

10.74 3.30.0 (2021-03-16)

10.74.1 New features

- config: read required variables from environment (#85)
- db-create: add drop option to control database deletion (#202) *BREAKING CHANGE*: cubicweb-ctl db-create no more drops the db in
- massive store: add an option to allow stores not to drop constraints (#219)
- pyramid-ctl: add “nb-threads” parameter to cubicweb-ctl pyramid (#119)
- urlpublish: add empty_rset_raises_404 flag on rql rewrite urls (#199)
- add script_attribute to add_js function (#210)
- cubicweb/cwconfig: authenticated SMTP outgoing email
- database/exception: include the query information in database error for better debugging
- upgrade Logilab’s dependencies to last versions
- web: only set “Vary: Accept-Language” when we translate something (#224)

10.74.2 Bug fixes

- rql2sql: properly handle date and datetime operations with SQLite (#109)
- rql: make the rql completion working again
- rql: refactor GROUP_CONCAT so that it handles NULL values
- catch authentication exception
- ci: manually remove the .tox/doc directory (#206)
- ci: use **/*.py to match all python files
- ci: recreate doc environment from scratch (#206)
- cwgettext: missing local module
- db-create: don’t force to use –drop if there is no existing db.
- deprecated: logilab.common.deprecated has been renamed to callable_deprecated
- deps: we are not yet compatible with pyramid 2.0
- migrations: don’t use notification hooks during postcreate
- py3: we still have some unicode() around
- repo_cnx: Catch OperationalError during repo_cnx (#215)
- skeleton: add gitlab-ci in skeleton manifest
- skeleton: make the skeleton black compliant
- skip a wdoc test when doctools is not available

- typo: drop_db instead of drop_pd
- urlpublisher: raise a 404 when a URL rewrite with rql has no rset (#199)
- UX when migrations failed to get its connection
- migrations: don't commit in the middle of drop_cube
- views: fix possible UnboundLocalError in ErrorView
- server: Set language of connection in all cases (#87)

10.74.3 Documentation

- deploy: add a Docker section in deployment
- deploy: Update kubernetes deploy
- deploy: Update uwsgi deployment
- deploy: add section ref for kubernetes section
- include api documentation
- mention weekly meeting in matrix
- Add link to migration and remove FIXME
- add more links in the index and capitalize entries (#185)
- all-in-one.conf: add link in index.rst
- dataimport: remove SQLGenObjectStore description and add MassiveObjectStore.
- index: remove “skeleton”, since it’s already explain in “layout”
- licence: automatically set licence info in setup.py template (#94)
- move (and fix) apache documentation to the deploy section
- Remove SQLSERVER
- rql: replace COMMA_JOIN by GROUP_CONCAT (#259)
- tutorials: correct a dead link.
- tutorials: add a link to museum demo source code, and correct a typo.
- tutorials: add introduction and structure of the museum tutorial.
- tutorials: redaction of “data-management/import” part of the museum tuto.
- tutorials: redaction of “getting started” part.
- tutorials: reword

10.74.4 Continuous integration

- uses gitlab-ci ‘rules:’
- integrate yamllint
- simplify rule:changes
- Use workflow to avoid duplicated pipelines (see <https://docs.gitlab.com/ee/ci/yaml/#switch-between-branch-pipelines-and-merge-request-pipelines>)
- do not run sonaube and deploy the doc when triggered by other project
- fix: also monitore requirements/setup.py changes for triggering the pipelines updated

10.74.5 Various changes

- remove statsd (closes #39) *BREAKING CHANGE*
- remove web.cors in favor of wsgicors with pyramid
- server/migrations: simplify the Migration Handler entry point
- server: replace utils.QueryCache with cachetools.LFUCache
- Silent yams warning (first rdef selection from an ambiguous rtype)
- Very minor improvements of cubicweb/server/repository.py
- views: Make JsonMixIn.wdata method usable with non-web connections

10.75 3.29.6 (2021-10-07)

10.75.1 New features

- Allow authentication on SMTP for outgoing emails. This feature was present in 3.28, but got lost during merge.

10.76 3.29

10.76.1 New features

- we started to reorganize the documentation, in particular the index, this work will continue with the next releases. Thus, some chapters are still missing
- ext/markdown: add urlize extension to auto link url in markdown documents

10.76.2 Bug fixes

- `.gitlab-ci.yml.tpl` should be named `.gitlab-ci.yml.tpl` for heptapod

10.76.3 Documentation

- Change data model link to use the definition from YAMS
- reorganize the TOC
- fix a few dead links
- add explanation on `site_cubicweb.py`
- update cube layout documentation
- add more explanations to what is a cube
- pooler: bad option name for the connections pooler
- Change data model link to use the definition from YAMS

10.76.4 Continuous integration

- build the documentation on gitlab pages

10.76.5 Various changes

- `DeprecatedWarning`: `[logilab.common.deprecation]` moved has been renamed and is deprecated, uses `callable_moved` instead
- add pipeline badge and shields with stats from pypi & docker

10.77 3.28.2

10.77.1 Fixed

- re-introduce `cubicweb.pyramid.resources.EntityResource/ETypeResource`
- re-introduce `cubicweb.pyramid.predicates.MatchIsETypePredicate`
- both were still needed for several cubes

10.78 3.28.1

10.78.1 Fixed

- python 3.8 compatibility: `base64 encodestring` has been removed, use `encodebytes`

10.79 3.28

The big highlights of this release are:

- CubicWeb now requires **python >=3.6**
- a new dynamic database connections pooler to replace the old static one
- a big upgrade in our CI workflow both for tests and documentations
- RDF generations when `rdm` mimetype in Accept HTTP headers
- `rql` resultset now stores selections variables names for RQL select queries, this will allow to build better tools

10.79.1 Added

- `[pyramid]`has_cw_permission`` pyramid predicates added for routes and view
- The database pooler is now dynamic. New connections are opened when needed and closed after a configurable period of time. This can be configured through *connections-pooler-max-size* (default 0, unlimited), *connections-pooler-min-size* (default 0), and *connections-pooler-idle-timeout* (default 600 seconds). The old configuration *connections-pooler-size* has been dropped.
- `[pyramid-debugtoolbar]` make SQL and RQL tables sortable
- `[RQL]Resultset` now stores selected variables for RQL select queries
- pyramid: add routes `/oid` and `/otype/oid` to return RDF when `rdm` mimetype in Accept HTTP headers
- entities: simplify `rdm` generation and add a generic `rdm` adapter
- `web.views`: add `Link` alternate in HTTP response header in HTML view
- Black and Mypy config in `tox.ini` file in new cube skeleton
- Gitlab CI config file in new cube skeleton

10.79.2 Changed

- CubicWeb now requires `python >=3.6`
- CI now test Cubicweb against latest unreleased public commits of its dependencies
- CI now rebuilds documentation after tests

10.79.3 Deprecated

- Class `cubicweb.view.EntityAdapter` was moved to `cubicweb.entity.EntityAdapter`, a deprecation warning is in place, but please update your source code accordingly

10.79.4 Removed

- Support for plpython has been dropped
- *RDFnquadsView* (**Breaking Change**)

10.79.5 Fixed

- Fix various tests in the CI
- Use SchemaLoader instead of pyfilereader
- [pyramid-debugtoolbar] remove CW controller panel rendering when no controller got collected
- [basecontroller] link tags in the header can only be added on entities
- add a `__contains__` method to `dict_protocol_catcher` to avoid breaking on “in”

Thanks to our contributors: Simon Chabot, Laurent Peuch, Nicolas Chauvat, Philippe Pepiot, Élodie Thieblin, François FERRY, Fabien Amarger, Laurent Wouters, Guillaume Vandeveld.

10.80 3.27 (31 January 2020)

10.80.1 New features

- Tests can now be run concurrently across multiple processes. You can use `pytest-xdist` for that. For tests using *PostgresApptestConfiguration* you should be aware that *startpgcluster()* can’t run concurrently. Workaround is to call `pytest` with `--dist=loadfile` to use a single test process per test module or use an existing database cluster and set `db-host` and `db-port` of `devtools.DEFAULT_PSQL_SOURCES['system']` accordingly.
- on *cubicweb-ctl create* and *cubicweb-ctl pyramid*, if it doesn’t already exist in the instance directory, the *pyramid.ini* file will be generated with the needed secrets.
- add a `-pdb` flag to all *cubicweb-ctl* command to launch (i)pdb if an exception occurs during a command execution.
- the `-loglevel` and `-dbglevel` flags are available for all *cubicweb-ctl* instance commands (and not only the *pyramid* one)
- following “only in foreground” behavior all commands logs to stdout by default from now on. To still log to a file pass `log_to_file=True` to `CubicWebConfiguration.config_for`
- add a new migration function *update_bfss_path(old_path, new_path)* to update the path in Bytes File-System Storage (bfss).
- on every request display request path and selected controller in CLI
- migration interactive mode improvements:
 - when an exception occurs, display the full traceback instead of only the exception
 - on migration p(db) choice, launch ipdb if it’s installed
 - on migration p(db) choice, give the traceback to pdb if it’s available, this mean that the (i)pdb interactive session will be on the stack of the exception instead of being on the stack where pdb is launched which will allow the user to access all the relevant context of the exception which otherwise is lost
- on `DBG_SQL` and/or `DBG_RQL`, if `pygments` is installed, syntax highlight sql/rql debug output
- allow to specify the instance id for any instance command using the `CW_INSTANCE` global variable instead of or giving it as a cli argument

- when debugmode is activated ('-D/-debug' on the pyramid command for example), the HTML generated by CW will contains new tags that will indicate by which object in the code it has been generated and in which line of which source code. For example:

```
<div
  cubicweb-generated-by="cubicweb.web.views.basetemplates.TheMainTemplate"
  cubicweb-from-source="/home/user/code/logilab/cubicweb/cubicweb/web/views/
↪basetemplates.py:161"
  id="contentmain">
  <h1
    cubicweb-generated-by="cubicweb.web.views.basetemplates.TheMainTemplate"
    cubicweb-from-source="/home/user/code/logilab/cubicweb/cubicweb/view.py:136">
    unset title
  </h1>
  [...]
</div>
```

While this hasn't been done yet, this feature is an open path for building dynamic tools that can help inspect the page.

- a new debug channels mechanism has been added, you can subscribe to one of those channels in your python code to build debug tools for example (the pyramid custom panels are built using that) and you will receive a datastructure (a dict) containing related information. The available channels are: controller, rql, sql, vreg, registry_decisions
- add a new '-t/-toolbar' option the pyramid command to activate the pyramid debugtoolbar
- a series of pyramid debugtoolbar panels specifically made for CW, see bellow

10.80.2 Pyramid debugtoolbar and custom panel

The pyramid debugtoolbar is now integrated into CubicWeb during the development phase when you use the 'pyramid' command. To activate it you need to pass the '-t/-toolbar' argument to the 'pyramid' command.

In addition, a series of custom panels specifically done for CW are now available, they display useful information for the development and the debugging of each page. The available panels are:

- a general panel which contains the selected controller, the current settings and useful links [screenshot](#)
- a panel listing all decisions taken in registry for building this page [screenshot](#)
- a panel listing the content of the vreg registries [screenshot](#)
- a panel listing all the RQL queries made during a request [screenshot](#)
- a panel listing all the SQL queries made during a request [screenshot](#)

Furthermore, in all those panels, next to each object/class/function/method a link to display its source code is available (shown as '[source]' [screenshot](#)) and also every file path shown in a traceback is also a link to display the corresponding file ([screenshot](#)). For example: [screenshot](#).

10.80.3 Backwards incompatible changes

- Standardization on the way to launch a cubicweb instance, from now on the only way to do that will be the used the `pyramid` command. Therefore:
 - `cubicweb-ctl` commands “start”, “stop”, “restart”, “reload” and “status” have been removed because they relied on the Twisted web server backend that is no longer maintained nor working with Python 3.
 - Twisted web server support has been removed.
 - `cubicweb-ctl wsgi` has also been removed.
- Support for legacy cubes (in the ‘cubes’ python namespace) has been dropped. Use of environment variables `CW_CUBES_PATH` and `CUBES_DIR` is removed.
- Python 2 support has been dropped.
- Exceptions in notification hooks aren’t caught anymore during tests so one can expect tests that seem to pass (but were actually silently failing) to fail now.
- All “cubicweb-ctl” command only accept one instance argument from now one (instead of 0 to n)
- ‘pyramid’ command will always run in the foreground now, by consequence the option `--no-daemon` has been removed.
- `DBG_MS` flag has been removed since it is not used anymore
- transactions db logs were displayed using the logging (debug/info/warning...) mechanism, now it is only displayed if the corresponding `DBG_OPS` flag is used
- backward python 2 compatible code for `scheduler` class has been removed from `cubicweb.server.utils`. If you get an import error when doing `from cubicweb.server.utils import scheduler` replace it with `from sched import scheduler`.

10.80.4 Deprecated code drops

Most code deprecated until version 3.25 has been dropped.

10.81 3.26 (1 February 2018)

10.81.1 New features

- For pyramid instance configuration kind, logging is not handled anymore by CubicWeb but should be configured through `development.ini` file following <https://docs.pylonsproject.org/projects/pyramid/en/latest/narr/logging.html>.

10.81.2 Backwards incompatible changes

- `CubicWebConfiguration` method ‘`shared_dir`’ got dropped.

10.82 3.25 (14 April 2017)

10.82.1 New features

- A new option *connections-pooler-enabled* (default yes) has been added. This allow to switch off internal connection pooling for use with others poolers such as [pgbouncer](#).
- In *deleteconf* view (confirmation before deletion), the list of first-level composite objects that would be deleted along with the primary entity is displayed (01eeea97e549).
- The `cubicweb.pyramid` module now provides a Paste application factory registered as an entry point named `pyramid_main` and that can be used to run a Pyramid WSGI application bound to a CubicWeb repository.
- A new configuration type `pyramid` has been added to create CubicWeb's instances (through `cubicweb-ctl create -c pyramid <basecube> <appid>`). This configuration bootstraps a CubicWeb instance that is essentially a repository plus the minimal setup to run a Pyramid WSGI application on top of it. Noticeably, it does not ship all *web* configuration but rather relies on configuration declared in a `development.ini` file for any Pyramid application.
- A new way to declare workflows as simple data structure (dict/list) has been introduced. Respective utility functions live in `cubicweb.wfutils` module. This handles both the creation and migration of workflows.
- A new IDublinCore adapter has been introduced to control the generation of Dublin Core metadata that are used in several base views.
- It is now possible to *derive* rtags using their `derive` method (0849a5eb57b8). Derived rtags keep a reference to the original rtag and only hold custom rules, allowing changes which are done in the original rtag after derivation to be still considered.
- A new `cubicweb-ctl scheduler <appid>` command has been introduced to run background and periodic tasks of the repository (previously called *looping tasks*). In a production environment, a process with this command should be run alongside with a WSGI server process (possibly running multiple processes itself).

10.82.2 Backwards incompatible changes

- As a consequence of the replacement of the old looping tasks manager by a scheduler, all `cubicweb-ctl`'s "start" commands (i.e. `start`, `pyramid`, `wsgi`) do not start repository *looping tasks manager* anymore, nor do they start the scheduler. Site administrators are thus expected to start this scheduler as a separate process. Also, registering looping tasks (i.e. calling `repo.looping_tasks()`) is a no-op when the repository has no scheduler set; a warning is issued in such cases. Application developers may rely on repository's `has_scheduler` method to determine if they should register a looping task or not.
- In `cubicweb.pyramid`, function `make_cubicweb_application` got renamed into `config_from_cwconfig` (950ce7d9f642).
- Several cleanups in repository's session management have been conducted resulting from changes introduced in 3.19 release. Among others, the `cubicweb.server.session.Session` class has been dropped, and request `session` attribute is now tight to a web session whose implementation depends on the front-end used (twisted or pyramid). Hence this attribute should not be accessed from "repository side" code (e.g. hooks or operations) and has lost some of his former attributes like `repo` which used to reference the repository instance. Due to this, you don't have anymore access to session's data through the connection, which leads to deprecation of the `data` attribute and removal of `get_shared_data` and `set_shared_data` methods which are deprecated since 3.19.
- Support for 'https-url' configuration option has been removed (4516c3956d46).
- The `next_tabindex` method of request class has been removed (011730a4af73). This include the removal of `settabindex` from the `FieldWidget` class init method.

- The *cubicweb.hook.logstats.start* hook was dropped because it's looping task would not be run in a web instance (see first point about repository scheduler).
- *uicfg* rules to hide the opposite relation of inlined form are not anymore automatically added, because this was actually done randomly and so not reliable, so you'll have to add them manually:

```
autoform_section.tag_subject_of(('CWUser', 'use_email', 'EmailAddress'),
                                'main', 'inlined')
autoform_section.tag_object_of(('CWUser', 'use_email', 'EmailAddress'),
                                'inlined', 'hidden')
```

10.83 3.24 (2 November 2016)

10.83.1 New features

- Various bits of a CubicWeb application configuration can be now be overridden through environments variables matching configuration option names prefixed by *CW_* (for instance *CW_BASE_URL*).
- Cubes are now standard Python packages named as *cubicweb-<cubeName>*. They are not anymore installed in *<prefix>/share/cubicweb/cubes*. Their discovery by CubicWeb is handled by a new setuptools entry point *cubicweb.cubes*. A backward compatibility layer is kept for “legacy” cubes.
- Pyramid support made it into CubicWeb core. Applications that use it should now declare the *cubicweb[pyramid]* dependency instead of *cubicweb-pyramid*.
- New *NullStore* class in *cubicweb.dataimport.stores* as new base class for every store, and allowing to test your dataimport chain without actually importing anything.

10.83.2 Major changes

There has been several important changes to the core internals of CubicWeb:

- Dropped *asource* and *extid* columns from the *entities* table as well as the index on the *type* column, for a sensible optimization on both massive data insertion and database size / index rebuilding.
- Dropped the *moved_entities* table and related mechanism to remember that an entity has been moved from a source to the system database - this is now the responsibility of source's parser to detect this (usually by remembering its original external id as *cwuri*).
- Dropped the original ‘give me an eid for this extid, but call me back on another line if it has to be created’ mechanism on which the *cxmlparser* was relying, in favor of parsers using the dataimport API. This includes dropping the *cxmlparser*. If you're using it, you'll have to write a specific parser, examples to come.
- Dropped source mapping handling (schema, views, logic) with its client the *cxmlparser*. This is not worth the burden, specific parsers should be preferred.

The above changes lead to the following API changes:

- *req.entity metas(eid)* doesn't return anymore a ‘type’ nor ‘source’ keys, use *req.entity_type(eid)* instead or ‘cw_source’ relation to get those information,
- deprecated *entity.cw_metainformation()*, which doesn't return anymore it's ‘source’ key,
- dropped *repository.type_and_source_from_eid(eid, cnx)*, *repository.extid2eid(...)* and *source.eid_type_source(cnx, eid)*,
- dropped *source.support_entity(etype)* and *source.support_relation(rtype)*,

- dropped 'cw_source' key from default JSON representation of an entity,
- dropped *source_uris()* and *handle_deletion(...)* method from datafeed parser base class, deletion of entities is now the responsibility of specific implementation (see *ldapparser* for example),
- entities from external source are always displayed in the UI with a link to the local entity, not the original one simplifying *entity.absolute_url()* implementation and allowing to drop *use_ext_eid* argument of *entity.rest_path()* (though it's still supported for backward compat).

10.83.3 Changes to the massive store

Several improvements have been done to `cubicweb.dataimport.massive_store`, with among the more important ones:

- Extended store API to provide more control to end-users: *fill_entities_table*, *fill_relation_table*, *fill_meta_relation_table*.
- Dropped *on_commit* / *on_rollback* arguments of the constructor.
- Use a slave specific temporary table for entities insertion as for relations (should improve concurrency when using in master/slaves mode).
- Delay dropping of constraint to the *finish* method, avoiding performance problem that was occurring because indexes were dropped at store creation time.
- Consider the given metadata generator when looking for which metadata tables should have their constraints dropped.
- Don't drop index on *entities.eid*, it's too costly to rebuild on database with some million of entities.

10.84 3.23 (24 June 2016)

10.84.1 New features

- Python 3.x support in CubicWeb itself is now complete, except for the twisted package (since Twisted does not completely support Python 3.x itself). The skeleton for new cube should also be Python 3 compatible, in particular its *setup.py* got updated.
- The *source-sync* command can now synchronize all sources in the database, if no *<source>* argument is provided.
- Datafeed source synchronization is now asynchronous when requested from user interface.
- Most indexes and constraints will be rebuilt during the migration, because they are now named after a md5 hash to control the name's size.
- Index are renamed upon renaming of an entity type, so they are still correctly tracked.
- A new *db-check-index* command is added to *cubicweb-ctl*, to display the differences between the indexes in the database and those expected by the schema. It's recommended to run this command after the migration to 3.23 and to adjust things manually for cases that are not easily handled by the migration script, such as indexes of entity types that have been renamed. It should be mostly about dropping extra indexes.
- Deprecated *MetaGenerator* in favor of slightly adapted API in *MetadataGenerator* (more consistent, giving more control to sub-classes and suitable for usage with the *MassiveObjectStore*)
- Major cleanups of the *MassiveObjectStore* and its *PGHelper* companion class:
 - dropped a bunch of unnecessary / unused attributes
 - refactored / renamed internal methods

- added support for a metadata generator, the now recommended way to control metadata generation
- Deprecated *SQLGenObjectStore*, *MassiveObjectStore* should be used instead.

10.84.2 Backwards-incompatible changes

- Generative tests *à la logilab-common* are not supported anymore in *CubicWebTC*. It is advised to use the [subtests](#) API (available on *CubicWebTC* either from the standard library as of Python 3.4 or through unittest2 package otherwise).
- *CubicWebTC*'s *set_description* method (comming from *logilab.common.testlib.TestCase*) is no longer available.

10.84.3 Development

When installed within a virtualenv, CubicWeb will look for instances data as in user mode by default, that is in `$HOME/etc/cubicweb.d`, as opposed to `$VIRTUAL_ENV/etc/cubicweb.d` previously. To restore this behavior, explicitly set `CW_MODE` to `system`. Alternatively (and preferably), the `CW_INSTANCES_DIR` environment variables may be used to specify instances data location.

10.85 3.22 (4 January 2016)

10.85.1 New features

- a huge amount of changes were done towards python 3.x support (as yet incomplete). This introduces a new dependency on `six`, to handle python2/python3 compatibility.
- new `cubicweb.dataimport.massive_store` module, a postgresql-specific store using the `COPY` statement to accelerate massive data imports. This functionality was previously part of `cubicweb-dataio` (there are some API differences with that previous version, however).
- cubes custom sql scripts are executed before creating tables. This allows them to create new types or extensions.
- the `ejsonexport` view can be specialized using the new `ISerializable` entity adapter. By default, it will return an entity's (non-Bytes and non-Password) attributes plus the special `cw_etype` and `cw_source` keys.
- cubes that define custom final types are now handled by the `add_cube` migration command.
- synchronization of external sources can be triggered from the web interface by suitably privileged users with a new `cw.source-sync` action.

10.85.2 User-visible changes

- the `ldapfeed` source now depends on the `ldap3` module instead of `python-ldap`.
- replies don't get an `Expires` header by default. However when they do, they also get a coherent `Cache-Control`.
- data files are regenerated at each request, they are no longer cached by `cubicweb.web.PropertySheet`. Requests for data files missing the instance hash are handled with a redirection instead of a direct reply, to allow correct cache-related reply headers.

10.85.3 API changes

- `config.repository()` creates a new `Repository` object each time, instead of returning a cached object. WARNING: this may cause unexpected issues if several repositories end up being used.
- migration scripts, as well as other scripts executed by `cubicweb-ctl shell`, are loaded with the `print_function` flag enabled (for backwards compatibility, if that fails they are re-loaded without that flag)
- the `cw_fti_index_rql_queries` method on entity classes is replaced by `cw_fti_index_rql_limit`, a generator which yields `ResultSet` objects containing entities to be indexed. By default, entities are returned 1000 at a time.
- `IDownloadableAdapter` API is clarified: `download_url`, `download_content_type` and `download_file_name` return unicode objects, `download_data` returns bytes.
- the `Repository.extid2eid()` entry point for external sources is deprecated. Imports should use one of the stores from the `cubicweb.dataimport` package instead.
- the `cubicweb.repoapi.get_repository()` function's `uri` argument should no longer be used.
- the generic datafeed xml parser is deprecated in favor of the "store" API introduced in cubicweb 3.21.
- the session manager lives in the `sessions` registry instead of `components`.
- `TZDatetime` attributes are returned as timezone-aware python datetime objects. WARNING: this will break client applications that compare or use arithmetic involving timezone-naive datetime objects.
- `creation_date` and `modification_date` attributes for all entities are now timezone-aware (`TZDatetime`) instead of `localtime` (`Datetime`). More generally, the `Datetime` type should be considered as deprecated.

10.85.4 Deprecated code drops

- the `cubicweb.server.hooksmanager` module was removed
- the `Repository.pinfo()` method was removed
- the `cubicweb.utils.SizeConstrainedList` class was removed
- the 'startorder' file in configuration directory is no longer honored

10.86 3.21 (10 July 2015)

10.86.1 New features

- the `datadir-url` configuration option lets one choose where static data files are served (instead of the default `${base-url}/data/`)
- some integrity checking that was previously implemented in Python was moved to the SQL backend. This includes some constraints, and referential integrity. Some consequences are that:
 - disabling integrity hooks no longer disables those checks
 - upgrades that modify constraints will fail when running on `sqlite` (but upgrades aren't supported on `sqlite` anyway)

Note: as of 3.21.0, the upgrade script only works on PostgreSQL. The migration for `SQLServer` will be added in a future bugfix release.

- for easier instance monitoring, cubicweb can regularly dump some statistics (basically those exposed by the 'info' and 'gc' views) in json format to a file

10.86.2 User-visible changes

- the use of fckeditor for text form fields is disabled by default, to re-enable it simply install the *cubicweb-ckeditor* cube (then `add_cude('ckeditor')` in a migration or in the shell)
- the 'https-deny-anonymous' configuration setting no longer exists

10.86.3 Code movement

The `cubicweb.web.views.timeline` module (providing the `timeline-json`, `timeline` and `static-timeline` views) has moved to a standalone [cube](#)

10.86.4 API changes

- `req.set_cookie`'s "expires" argument, if not `None`, is expected to be a date or a datetime in UTC. It was previously interpreted as localtime with the UTC offset the server started in, which was inconsistent (we are not aware of any users of that API).
- the way to run tests on a postgresql backend has changed slightly, use `cubicweb.devtools.{start,stop}pgcluster` in `setUpModule` and `tearDownModule`
- the `Connection` and `ClientConnection` objects introduced in CubicWeb 3.19 have been unified. To connect to a repository, use:

```
session = repo.new_session(login, password=...)
with session.new_cnx() as cnx:
    cnx.execute(...)
```

In tests, the `'repo_cnx'` and `'client_cnx'` methods of `RepoAccess` are now aliases to `'cnx'`.

10.86.5 Deprecated code drops

- the `user_callback` api has been removed; people should use plain ajax functions instead
- the *Pyro* and *Zmq-pickle* remote repository access methods have been entirely removed (emerging alternatives such as `rqcontroller` and `cwclientlib` should be used instead). Note that as a side effect, "repository-only" instances (i.e. without a http component) are no longer possible. If you have any such instances, you will need to rename the configuration file from `repository.conf` to `all-in-one.conf` and run `cubicweb-ctl upgrade` to update it. Likewise, remote `cubicweb-ctl` shell is no longer available.
- the old (deprecated since 3.19) *DBAPI* api is completely removed
- `cubicweb.toolsutils.config_connect()` has been removed

10.87 3.20 (06/01/2015)

10.87.1 New features

- virtual relations: a new `ComputedRelation` class can be used in `schema.py`; its *rule* attribute is an RQL snippet that defines the new relation.
- computed attributes: an attribute can now be defined with a *formula* argument (also an RQL snippet); it will be read-only, and updated automatically.

Both of these features are described in [CWEP-002](#), and the updated “Data model” chapter of the CubicWeb book.

- cubicweb-ctl plugins can use the `cubicweb.utils.adminctx` function to get a `Connection` object from an instance name.
- new ‘tornado’ wsgi backend
- session cookies have the `HttpOnly` flag, so they’re no longer exposed to javascript
- rich text fields can be formatted as markdown
- the edit controller detects concurrent editions, and raises a `ValidationError` if an entity was modified between form generation and submission
- cubicweb can use a postgresql “schema” (namespace) for its tables
- “cubicweb-ctl configure” can be used to set values of the admin user credentials in the sources configuration file
- in debug mode, setting the `_cwtracehtml` parameter on a request allows tracing where each bit of output is produced

10.87.2 API Changes

- `ucsvreader()` and `ucsvreader_pb()` from the `dataimport` module have 2 new keyword arguments `delimiter` and `quotechar` to replace the `separator` and `quote` arguments respectively. This makes the API match that of Python’s `csv.reader()`. The old arguments are still supported though deprecated.
- the migration environment’s `remove_cube` function is now called `drop_cube`.
- `cubicweb.old.css` is now `cubicweb.css`. The previous “new” `cubicweb.css`, along with its `cubicweb.reset.css` companion, have been removed.
- the jquery-treeview plugin was updated to its latest version

10.87.3 Deprecated Code Drops

- most of 3.10 and 3.11 backward compat is gone; this includes:
 - `CtxComponent.box_action()` and `CtxComponent.build_link()`
 - `cubicweb.devtools.htmlparser.XMLDemotingValidator`
 - various methods and properties on `Entities`, replaced by `cw_edited` and `cw_attr_cache`
 - ‘`commit_event`’ method on `hooks`, replaced by ‘`postcommit_event`’
 - `server.hook.set_operation()`, replaced by `Operation.get_instance(...).add_data()`
 - `View.div_id()`, `View.div_class()` and `View.create_url()`
 - **VComponent* classes

- in forms, `Field.value()` and `Field.help()` must take the form and the field itself as arguments
- `form.render()` must get `w` as a named argument, and `renderer.render()` must take `w` as first argument
- in breadcrumbs, the optional `recurs` argument must be a set, not `False`
- `cubicweb.web.views.idownloadable`.{`download_box`, `IDownloadableLineView`}
- primary views no longer have `render_entity_summary` and `summary` methods
- `WFHistoryVComponent`'s `cell_call` method is replaced by `render_body`
- `cubicweb.dataimport.ObjectStore.add()`, replaced by `create_entity`
- `ManageView`.{`folders`, `display_folders`}

10.88 3.19 (28/04/2015)

10.88.1 New functionalities

- implement Cross Origin Resource Sharing (CORS) (see [#2491768](#))
- `system_source.create_eid` can get a range of IDs, to reduce overhead of batch entity creation

10.88.2 Behaviour Changes

- The anonymous property of `Session` and `Connection` are now computed from the related user login. If it matches the `anonymous-user` in the config the connection is anonymous. Beware that the `anonymous-user` config is web specific. Therefore, no session may be anonymous in a repository only setup.

10.88.3 New Repository Access API

A new explicit `Connection` object replaces `Session` as the main repository entry point. `Connection` holds all the necessary methods to be used server-side (`execute`, `commit`, `rollback`, `call_service`, `entity_from_eid`, etc...). One obtains a new `Connection` object using `session.new_cnx()`. `Connection` objects need to have an explicit begin and end. Use them as a context manager to never miss an end:

```
with session.new_cnx() as cnx:
    cnx.execute('INSERT Elephant E, E name "Babar"')
    cnx.commit()
    cnx.execute('INSERT Elephant E, E name "Celeste"')
    cnx.commit()
# Once you get out of the "with" clause, the connection is closed.
```

Using the same `Connection` object in multiple threads will give you access to the same `Transaction`. However, `Connection` objects are not thread safe (hence at your own risks).

`repository.internal_session` is deprecated in favor of `repository.internal_cnx`. Note that internal connections are now *safe* by default, i.e. the integrity hooks are enabled.

Backward compatibility is preserved on `Session`.

A new API has been introduced to replace the `dbapi`. It is called *repoapi*.

There are three relevant functions for now:

- `repoapi.get_repository` returns a `Repository` object either from an URI when used as `repoapi.get_repository(uri)` or from a config when used as `repoapi.get_repository(config=config)`.
- `repoapi.connect(repo, login, **credentials)` returns a `ClientConnection` associated with the user identified by the credentials. The `ClientConnection` is associated with its own `Session` that is closed when the `ClientConnection` is closed. A `ClientConnection` is a `Connection`-like object to be used client side.
- `repoapi.anonymous_cnx(repo)` returns a `ClientConnection` associated with the anonymous user if described in the config.

On the client/web side, the `Request` is now using a `repoapi.ClientConnection` instead of a `dbapi.connection`. The `ClientConnection` has multiple backward compatible methods to make it look like a `dbapi.Cursor` and `dbapi.Connection`.

`Session` used on the Web side are now the same than the one used Server side. Some backward compatibility methods have been installed on the server side `Session` to ease the transition.

The authentication stack has been altered to use the `repoapi` instead of the `dbapi`. Cubes adding new element to this stack are likely to break.

`Session` data can be accessed using the `cnx.data` dictionary, while transaction data is available through `cnx.transaction_data`. These replace the `[gs]et_shared_data` methods with optional `txid` kwarg.

All current methods and attributes used to access the repo on CubicWebTC are deprecated. You may now use a `RepoAccess` object. A `RepoAccess` object is linked to a new `Session` for a specified user. It is able to create `Connection`, `ClientConnection` and web side requests linked to this session:

```
access = self.new_access('babar') # create a new RepoAccess for user babar
with access.repo_cnx() as cnx:
    # some work with server side cnx
    cnx.execute(...)
    cnx.commit()
    cnx.execute(...)
    cnx.commit()

with access.client_cnx() as cnx:
    # some work with client side cnx
    cnx.execute(...)
    cnx.commit()

with access.web_request(elephant='babar') as req:
    # some work with client side cnx
    elephant_name = req.form['elephant']
    req.execute(...)
    req.cnx.commit()
```

By default `testcase.admin_access` contains a `RepoAccess` object for the default admin session.

10.88.4 API changes

- `RepositorySessionManager.postlogin` is now called with two arguments, request and session. And this now happens before the session is linked to the request.
- `SessionManager` and `AuthenticationManager` now take a repo object at initialization time instead of a vreg.
- The `async` argument of `_cw.call_service` has been dropped. All calls are now synchronous. The `zmq` notification bus looks like a good replacement for most async use cases.
- `repo.stats()` is now deprecated. The same information is available through a service (`_cw.call_service('repo_stats')`).
- `repo.gc_stats()` is now deprecated. The same information is available through a service (`_cw.call_service('repo_gc_stats')`).
- `repo.register_user()` is now deprecated. The functionality is now available through a service (`_cw.call_service('register_user')`).
- `request.set_session` no longer takes an optional `user` argument.
- CubicwebTC does not have `repo` and `cnx` as class attributes anymore. They are standard instance attributes. `set_cnx` and `_init_repo` class methods become instance methods.
- `set_cnxset` and `free_cnxset` are deprecated. `cnxset` are now automatically managed.
- The implementation of cascading deletion when deleting *composite* entities has changed. There comes a semantic change: merely deleting a composite relation does not entail any more the deletion of the component side of the relation.
- `_cw.user_callback` and `_cw.user_rql_callback` are deprecated. Users are encouraged to write an actual controller (e.g. using `ajaxfunc`) instead of storing a closure in the session data.
- A new entity `entity.cw_linkable_rql` method provides the `rql` to fetch all entities that are already or may be related to the current entity using the given relation.

10.88.5 Deprecated Code Drops

- `session.hijack_user` mechanism has been dropped.
- `EtypeRestrictionComponent` has been removed, its functionality has been replaced by facets a while ago.
- the old multi-source support has been removed. Only copy-based sources remain, such as `datafeed` or `ldapfeed`.

10.89 3.18 (10/01/2014)

The migration script does not handle `sqlite` nor `mysql` instances.

10.89.1 New functionalities

- add a security debugging tool (see [#2920304](#))
- introduce an *add* permission on attributes, to be interpreted at entity creation time only and allow the implementation of complex *update* rules that don't block entity creation (before that the *update* attribute permission was interpreted at entity creation and update time)
- the primary view display controller (uicfg) now has a *set_fields_order* method similar to the one available for forms
- new method *ResultSet.one(col=0)* to retrieve a single entity and enforce the result has only one row (see [#3352314](#))
- new method *RequestSessionBase.find* to look for entities (see [#3361290](#))
- the embedded jQuery copy has been updated to version 1.10.2, and jQuery UI to version 1.10.3.
- initial support for wsgi for the debug mode, available through the new `wsgi cubicweb-ctl` command, which can use either python's builtin wsgi server or the werkzeug module if present.
- a `rql-table` directive is now available in ReST fields
- cubicweb-ctl upgrade can now generate the static data resource directory directly, without a manual call to `gen-static-datadir`.

10.89.2 API changes

- not really an API change, but the entity permission checks are now systematically deferred to an operation, instead of a) trying in a hook and b) if it failed, retrying later in an operation
- The default value storage for attributes is no longer String, but Bytes. This opens the road to storing arbitrary python objects, e.g. numpy arrays, and fixes a bug where default values whose truth value was False were not properly migrated.
- *symmetric* relations are no more handled by an rql rewrite but are now handled with hooks (from the *activeintegrity* category); this may have some consequences for applications that do low-level database manipulations or at times disable (some) hooks.
- *unique together* constraints (multi-columns unicity constraints) get a *name* attribute that maps the CubicWeb constraint entities to corresponding backend index.
- `BreadCrumbEntityVComponent`'s `open_breadcrumbs` method now includes the first breadcrumbs separator
- entities can be compared for equality and hashed
- the `on_fire_transition` predicate accepts a sequence of possible transition names
- the `GROUP_CONCAT` rql aggregate function no longer repeats duplicate values, on the sqlite and postgresql backends

10.89.3 Deprecation

- `pyrqrql` sources have been deprecated. Multisource will be fully dropped in the next version. If you are still using `pyrqrql`, switch to `datafeed` **NOW**!
- the old multi-source system
- *find_one_entity* and *find_entities* in favor of *find* (see [#3361290](#))
- the *TmpFileViewMixin* and *TmpPngView* classes (see [#3400448](#))

10.89.4 Deprecated Code Drops

- `ldapuser` have been dropped; use `ldapfeed` now (see [#2936496](#))
- action `GotRhythm` was removed, make sure you do not import it in your cubes (even to unregister it) (see [#3093362](#))
- all 3.8 backward compat is gone
- all 3.9 backward compat (including the javascript side) is gone
- the `twisted` (web-only) instance type has been removed

10.90 3.17 (02/05/2013)

10.90.1 New functionalities

- add a command to compare db schema and file system schema (see [#464991](#))
- Add `CubicWebRequestBase.content` with the content of the HTTP request (see [#2742453](#)) (see [#2742453](#))
- Add directive `bookmark` to ReST rendering (see [#2545595](#))
- Allow user defined final type (see [#124342](#))

10.90.2 API changes

- drop `typed_eid()` in favour of `int()` (see [#2742462](#))
- The SIOC views and adapters have been removed from CubicWeb and moved to the *sioc* cube.
- The web page embedding views and adapters have been removed from CubicWeb and moved to the *embed* cube.
- The email sending views and controllers have been removed from CubicWeb and moved to the *massmailing* cube.
- `RenderAndSendNotificationView` is deprecated in favor of `ActualNotificationOp` the new operation use the more efficient *data* idiom.
- Looping task can now have a interval ≤ 0 . Negative interval disable the looping task entirely.
- We now serve html instead of xhtml. (see [#2065651](#))

10.90.3 Deprecation

- `ldapuser` have been deprecated. It'll be fully dropped in the next version. If you are still using `ldapuser` switch to `ldapfeed` **NOW**!
- `hijack_user` have been deprecated. It will be dropped soon.

10.90.4 Deprecated Code Drops

- The progress views and adapters have been removed from CubicWeb. These classes were deprecated since 3.14.0. They are still available in the *iprogress* cube.
- API deprecated since 3.7 have been dropped.

10.91 3.16 (25/01/2013)

10.91.1 New functionalities

- Add a new dataimport store (*SQLGenObjectStore*). This store enables a fast import of data (entity creation, link creation) in CubicWeb, by directly flushing information in SQL. This may only be used with PostgreSQL, as it requires the 'COPY FROM' command.

10.91.2 API changes

- Orm: *set_attributes* and *set_relations* are unified (and deprecated) in favor of *cw_set* that works in all cases.
- db-api/configuration: all the external repository connection information is now in an URL (see [#2521848](#)), allowing to drop specific options of pyro nameserver host, group, etc and fix broken ZMQ source. Configuration related changes:
 - Dropped 'pyro-ns-host', 'pyro-instance-id', 'pyro-ns-group' from the client side configuration, in favor of 'repository-uri'. **NO MIGRATION IS DONE**, supposing there is no web-only configuration in the wild.
 - Stop discovering the connection method through *repo_method* class attribute of the configuration, varying according to the configuration class. This is a first step on the way to a simpler configuration handling.

DB-API related changes:

- Stop indicating the connection method using *ConnectionProperties*.
- Drop *_cnxtype* attribute from *Connection* and *cnxtype* from *Session*. The former is replaced by a *is_repo_in_memory* property and the later is totally useless.
- Turn *repo_connect* into *_repo_connect* to mark it as a private function.
- Deprecate *in_memory_cnx* which becomes useless, use *_repo_connect* instead if necessary.
- the "tcp://" uri scheme used for ZMQ communications (in a way reminiscent of Pyro) is now named "zmqpickle-tcp://", so as to make room for future zmq-based lightweight communications (without python objects pickling).
- Request.base_url gets a *secure=True* optional parameter that yields an https url if possible, allowing hook-generated content to send secure urls (e.g. when sending mail notifications)
- Dataimport ucsvreader gets a new boolean *ignore_errors* parameter.

10.91.3 Unintrusive API changes

- Drop of *cubicweb.web.uicfg.AutoformSectionRelationTags.bw_tag_map*, deprecated since 3.6.

10.91.4 User interface changes

- The RQL search bar has now some auto-completion support. It means relation types or entity types can be suggested while typing. It is an awesome improvement over the current behaviour !
- The *action box* associated with *table* views (from *tableview.py*) has been transformed into a nice-looking series of small tabs; it means that the possible actions are immediately visible and need not be discovered by clicking on an almost invisible icon on the upper right.
- The *uicfg* module has moved to *web/views/* and ui configuration objects are now selectable. This will reduce the amount of subclassing and whole methods replacement usually needed to customize the ui behaviour in many cases.
- Remove changelog view, as neither cubicweb nor known cubes/applications were properly feeding related files.

10.91.5 Other changes

- ‘pyrorql’ sources will be automatically updated to use an URL to locate the source rather than configuration option. ‘zmqrql’ sources were broken before this change, so no upgrade is needed...
- Debugging filters for Hooks and Operations have been added.
- Some cubicweb-ctl commands used to show the output of *msgcat* and *msgfmt*; they don’t anymore.

10.92 3.15 (12/04/2012)

10.92.1 New fonctionnalités

- Add Zmq server, based on the cutting edge ZMQ (<http://www.zeromq.org/>) socket library. This allows to access distant instance, in a similar way as Pyro.
- Publish/subscribe mechanism using ZMQ for communication among cubicweb instances. The new *zmq-address-sub* and *zmq-address-pub* configuration variables define where this communication occurs. As of this release this mechanism is used for entity cache invalidation.
- Improved WSGI support. While there is still some caveats, most of the code which was twisted only is now generic and allows related functionalities to work with a WSGI front-end.
- Full undo/transaction support : undo of modification has eventually been implemented, and the configuration simplified (basically you activate it or not on an instance basis).
- Controlling HTTP status code used is not much more easier :
 - *WebRequest* now has a *status_out* attribut to control the response status ;
 - most web-side exceptions take an optional *status* argument.

10.92.2 API changes

- The base registry implementation has been moved to a new *logilab.common.registry* module (see #1916014). This includes code from :

- *cubicweb.vreg* (the whole things that was in there)
- *cw.appobject* (base selectors and all).

In the process, some renaming was done:

- the top level registry is now *RegistryStore* (was *VRegistry*), but that should not impact cubicweb client code ;
- former selectors functions are now known as “predicate”, though you still use predicates to build an object’s selector ;
- for consistency, the *objectify_selector* decorator has hence be renamed to *objectify_predicate* ;
- on the CubicWeb side, the *selectors* module has been renamed to *predicates*.

Debugging refactoring dropped the more need for the *lltrace* decorator. There should be full backward compat with proper deprecation warnings. Notice the *yes* predicate and *objectify_predicate* decorator, as well as the *traced_selection* function should now be imported from the *logilab.common.registry* module.

- All login forms are now submitted to <app_root>/login. Redirection to requested page is now handled by the login controller (it was previously handle by the session manager).
- *Publisher.publish* has been renamed to *Publisher.handle_request*. This method now contains generic version of logic previously handled by Twisted. *Controller.publish* is **not** affected.

10.92.3 Unintrusive API changes

- New ‘ldapfeed’ source type, designed to replace ‘ldapuser’ source with data-feed (i.e. copy based) source ideas.
- New ‘zmqrql’ source type, similar to ‘pyrqrql’ but using ømq instead of Pyro.
- A new registry called *services* has appeared, where you can register server-side *cubicweb.server.Service* child classes. Their *call* method can be invoked from a web-side AppObject instance using new *self._cw.call_service* method or a server-side one using *self.session.call_service*. This is a new way to call server-side methods, much cleaner than monkey patching the Repository class, which becomes a deprecated way to perform similar tasks.
- a new *ajax-func* registry now hosts all remote functions (i.e. functions callable through the *asyncRemoteExec* JS api). A convenience *ajaxfunc* decorator will let you expose your python function easily without all the appobject standard boilerplate. Backward compatibility is preserved.
- the ‘json’ controller is now deprecated in favor of the ‘ajax’ one.
- *WebRequest.build_url* can now take a `__secure__` argument. When True cubicweb try to generate an https url.

10.92.4 User interface changes

A new ‘undohistory’ view expose the undoable transactions and give access to undo some of them.

10.93 3.14 (09/11/2011)

First notice CW 3.14 depends on yams 0.34 (which is incompatible with prior cubicweb releases regarding instance re-creation).

10.93.1 API changes

- *Entity.fetch_rql restriction* argument has been deprecated and should be replaced with a call to the new *Entity.fetch_rqlst* method, get the returned value (a rql *Select* node) and use the RQL syntax tree API to include the above-mentioned restrictions.

Backward compat is kept with proper warning.

- *Entity.fetch_order* and *Entity.fetch_unrelated_order* class methods have been replaced by *Entity.cw_fetch_order* and *Entity.cw_fetch_unrelated_order* with a different prototype:
 - instead of taking (attr, var) as two string argument, they now take (select, attr, var) where select is the rql syntax tree beinx constructed and var the variable *node*.
 - instead of returning some string to be inserted in the ORDERBY clause, it has to modify the syntax tree

Backward compat is kept with proper warning, BESIDE cases below:

- custom order method return **something else the a variable name with or without the sorting order** (e.g. cases where you sort on the value of a registered procedure as it was done in the tracker for instance). In such case, an error is logged telling that this sorting is ignored until API upgrade.
- client code use direct access to one of those methods on an entity (no code known to do that).
- *Entity._rest_attr_info* class method has been renamed to *Entity.cw_rest_attr_info*

No backward compat yet since this is a protected method an no code is known to use it outside cubicweb itself.

- *AnyEntity.linked_to* has been removed as part of a refactoring of this functionality (link a entity to another one at creation step). It was replaced by a *EntityFieldsForm.linked_to* property.

In the same refactoring, *cubicweb.web.formfield.relvoc_linkedto*, *cubicweb.web.formfield.relvoc_init* and *cubicweb.web.formfield.relvoc_unrelated* were removed and replaced by *RelationField* methods with the same names, that take a form as a parameter.

No backward compatibility yet. It’s still time to cry for it. Cubes known to be affected: tracker, vcsfile, vcreview.

- *CWPermission* entity type and its associated *require_permission* relation type (abstract) and *require_group* relation definitions have been moved to a new *localperms* cube. With this have gone some functions from the *cubicweb.schemas* package as well as some views. This makes cubicweb itself smaller while you get all the local permissions stuff into a single, documented, place.

Backward compat is kept for existing instances, **though you should have installed the localperms cubes**. A proper error should be displayed when trying to migrate to 3.14 an instance the use *CWPermission* without the new cube installed. For new instances / test, you should add a dependancy on the new cube in cubes using this feature, along with a dependancy on cubicweb >= 3.14.

- jQuery has been updated to 1.6.4 and jquery-tablesorter to 2.0.5. No backward compat issue known.

- Table views refactoring : new *RsetTableView* and *EntityTableView*, as well as rewritten an enhanced version of *PyValTableView* on the same bases, with logic moved to some column renderers and a layout. Those should be well documented and deprecates former *TableView*, *EntityAttributesTableView* and *CellView*, which are however kept for backward compat, with some warnings that may not be very clear unfortunately (you may see your own table view subclass name here, which doesn't make the problem that clear). Notice that `_cw.view('table', rset, *kwargs)` will be routed to the new *RsetTableView* or to the old *TableView* depending on given extra arguments. See #1986413.
- *display_name* don't call `.lower()` anymore. This may leads to changes in your user interface. Different msgid for upper/lower cases version of entity type names, as this is the only proper way to handle this with some languages.
- *IEditControlAdapter* has been deprecated in favor of *EditController* overloading, which was made easier by adding dedicated selectors called *match_edited_type* and *match_form_id*.
- Pre 3.6 API backward compat has been dropped, though *data* migration compatibility has been kept. You may have to fix errors due to old API usage for your instance before to be able to run migration, but then you should be able to upgrade even a pre 3.6 database.
- Deprecated *cubicweb.web.views.iprogress* in favor of new *iprogress* cube.
- Deprecated *cubicweb.web.views.flot* in favor of new *jqplot* cube.

10.93.2 Unintrusive API changes

- Refactored properties forms (eg user preferences and site wide properties) as well as pagination components to ease overriding.
- New *cubicweb.web.uihelper* module with high-level helpers for uicfg.
- New *anonymized_request* decorator to temporary run stuff as an anonymous user, whatever the currently logged in user.
- New 'verbatimattr' attribute view.
- New facet and form widget for Integer used to store binary mask.
- New *js_href* function to generated proper javascript href.
- *match_kwargs* and *match_form_params* selectors both accept a new *once_is_enough* argument.
- ***printable_value* is now a method of request, and may be given dict of formatters to use.**
- *[Rset]TableView* allows to set None in 'headers', meaning the label should be fetched from the result set as done by default.
- Field vocabulary computation on entity creation now takes `__linkto` information into account.
- Started a *cubicweb.pylintext* pylint plugin to help pylint analyzing cubes.

10.93.3 RQL

- Support for HAVING in 'SET' and 'DELETE' queries.
- new *AT_TZ* function to get back a timestamp at a given time-zone.
- new *WEEKDAY* date extraction function

10.93.4 User interface changes

- Datafeed source now present an history of the latest import's log, including global status and debug/info/warning/error messages issued during imports. Import logs older than a configurable amount of time are automatically deleted.
- Breadcrumbs component is properly kept when creating an entity with '___linkto'.
- users and groups management now really lead to that (i.e. includes *groups* management).
- New 'jsonp' controller with 'jsonexport' and 'ejsonexport' views.

10.93.5 Configuration

- Added option 'resources-concat' to make javascript/css files concatenation optional.

11.1 cubicweb

CubicWeb is a generic framework to quickly build applications which describes relations between entities.

11.1.1 Exceptions

Base exceptions

exception `cubicweb.ProgrammingError`

Bases: `Exception`

Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

exception `cubicweb.CubicWebException`

Bases: `Exception`

base class for cubicweb server exception

exception `cubicweb.InternalError`

Bases: `cubicweb._exceptions.CubicWebException`

base class for exceptions which should not occur

exception `cubicweb.SecurityError`

Bases: `cubicweb._exceptions.CubicWebException`

base class for cubicweb server security exceptions

exception `cubicweb.RepositoryError`

Bases: `cubicweb._exceptions.CubicWebException`

base class for repository exceptions

exception `cubicweb.SourceException`

Bases: `cubicweb._exceptions.CubicWebException`

base class for source exceptions

exception `cubicweb.CubicWebRuntimeError`

Bases: `cubicweb._exceptions.CubicWebException`

base class for runtime exceptions

Repository exceptions

exception `cubicweb.ConnectionError`

Bases: `cubicweb._exceptions.RepositoryError`

raised when a bad connection id is given or when an attempt to establish a connection failed

exception `cubicweb.AuthenticationError`

Bases: `cubicweb._exceptions.ConnectionError`

raised when an attempt to establish a connection failed due to wrong connection information (login / password or other authentication token)

exception `cubicweb.BadConnectionId`

Bases: `cubicweb._exceptions.ConnectionError`

raised when a bad connection id is given

exception `cubicweb.UnknownEid`

Bases: `cubicweb._exceptions.RepositoryError`

the eid is not defined in the system tables

exception `cubicweb.UniqueTogetherError`(*session*, ***kwargs*)

Bases: `cubicweb._exceptions.RepositoryError`

raised when a unique_together constraint caused an IntegrityError

Security Exceptions

exception `cubicweb.Unauthorized`

Bases: `cubicweb._exceptions.SecurityError`

raised when a user tries to perform an action without sufficient credentials

exception `cubicweb.Forbidden`

Bases: `cubicweb._exceptions.SecurityError`

raised when a user tries to perform a forbidden action

Source exceptions

exception `cubicweb.EidNotInSource`

Bases: `cubicweb._exceptions.SourceException`

trying to access an object with a particular eid from a particular source has failed

Registry exceptions

exception `cubicweb.UnknownProperty`

Bases: `logilab.common.registry.RegistryException`

property found in database but unknown in registry

Query exceptions

exception cubicweb.QueryError

Bases: `cubicweb._exceptions.CubicWebRuntimeError`

a query try to do something it shouldn't

exception cubicweb.NotAnEntity

Bases: `cubicweb._exceptions.CubicWebRuntimeError`

raised when `get_entity` is called for a column which doesn't contain a non final entity

exception cubicweb.MultipleResultsError

Bases: `cubicweb._exceptions.CubicWebRuntimeError`

raised when `ResultSet.one()` is called on a resultset with multiple rows of multiple columns.

exception cubicweb.NoResultError

Bases: `cubicweb._exceptions.CubicWebRuntimeError`

raised when no result is found but at least one is expected.

exception cubicweb.UndoTransactionException(*txuuid, errors*)

Bases: `cubicweb._exceptions.QueryError`

Raised when undoing a transaction could not be performed completely.

Note that :

- 1) the partial undo operation might be acceptable depending upon the final application
- 2) the undo operation can also fail with a *ValidationError* in cases where the undoing breaks integrity constraints checked immediately.
- 3) It might be that neither of those exception is raised but a subsequent *commit* might raise a *ValidationError* in cases where the undoing breaks integrity constraints checked at commit time.

Parameters

- **txuuid** – Unique identifier of the partially undone transaction
- **errors** (*list*) – List of errors occurred during undoing

Misc

exception cubicweb.ConfigurationError

Bases: `cubicweb._exceptions.CubicWebException`

a misconfiguration error

exception cubicweb.ExecutionError

Bases: `Exception`

server execution control error (already started, not running...)

exception cubicweb.BadCommandUsage

Bases: `Exception`

Raised when an unknown command is used or when a command is not correctly used (bad options, too much / missing arguments...).

Trigger display of command usage.

exception `cubicweb.ValidationError`(*entity*, *errors*: Dict, *msgargs*: Optional[Dict] = None, *i18nvalues*: Optional[List] = None)

Bases: `yams._exceptions.SchemaError`

Validation error details the reason(s) why the validation failed.

Arguments are:

- *entity*: the entity that could not be validated; actual type depends on the client library
- *errors*: errors dictionary, None key used for global error, other keys should be attribute/relation of the entity, qualified as subject/object using `yams.role_name()`. Values are the message associated to the keys, and may include interpolation string starting with '%(KEY-' where 'KEY' will be replaced by the associated key once the message has been translated. This allows predictable/translatable message and avoid args conflict if used for several keys.
- *msgargs*: dictionary of substitutions to be inserted in error messages once translated (only if *msgargs* is given)
- *i18nvalues*: list of keys in *msgargs* whose value should be translated

Translation will be done **in-place** by calling `translate()`.

11.1.2 Utilities

class `cubicweb.Binary`(*buf=b''*)

class to hold binary data. Use BytesIO to prevent use of unicode data

class `cubicweb.CubicWebEventManager`

simple event / callback manager.

Typical usage to register a callback:

```
>>> from cubicweb import CW_EVENT_MANAGER
>>> CW_EVENT_MANAGER.bind('after-registry-reload', mycallback)
```

Typical usage to emit an event:

```
>>> from cubicweb import CW_EVENT_MANAGER
>>> CW_EVENT_MANAGER.emit('after-registry-reload')
```

`emit()` accepts an additional context parameter that will be passed to the callback if specified (and only in that case)

`cubicweb.onevent`(*event*, **args*, ***kwargs*)

decorator to ease event / callback binding

```
>>> from cubicweb import onevent
>>> @onevent('before-registry-reload')
... def mycallback():
...     print 'hello'
...
>>>
```

`cubicweb.validation_error`(*entity*, *errors*, *substitutions*=None, *i18nvalues*=None)

easy way to retrieve a `cubicweb.ValidationError` for an entity or eid.

You may also have 2-tuple as error keys, `yams.role_name()` will be called automatically for them.

Messages in errors **should not be translated yet**, though marked for internationalization. You may give an additional substitution dictionary that will be used for interpolation after the translation.

11.2 cubicweb.appobject

11.2.1 The *AppObject* class

The AppObject class is the base class for all dynamically loaded objects (application objects) accessible through the registry.

We can find a certain number of attributes and methods defined in this class and common to all the application objects.

class cubicweb.appobject.**AppObject**(*req*, ****extra**)

Bases: logilab.common.registry.RegistrableObject

This is the base class for CubicWeb application objects which are selected in a request context.

The following attributes should be set on concrete appobject classes:

At selection time, the following attributes are set on the instance:

_cw current request

cw_extra_kwargs other received arguments

And also the following, only if *rset* is found in arguments (in which case *rset*/*row*/*col* will be removed from *cwextra_kwargs*):

cw_rset context result set or None

cw_row if a result set is set and the context is about a particular cell in the result set, and not the result set as a whole, specify the row number we are interested in, else None

cw_col if a result set is set and the context is about a particular cell in the result set, and not the result set as a whole, specify the col number we are interested in, else None

Note:

- do not inherit directly from this class but from a more specific class such as *AnyEntity*, *EntityView*, *AnyRsetView*, *Action*...
-

critical(*msg*, **args*, ****kwargs**)

Log ‘msg % args’ with severity ‘CRITICAL’.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

logger.critical(“Houston, we have a %s”, “major disaster”, *exc_info*=1)

cw_propval(*propid*)

return cw property value associated to key

<cls.__registry__>.<cls.id>.<propid>

debug(*msg*, **args*, ****kwargs**)

Log ‘msg % args’ with severity ‘DEBUG’.

To pass exception information, use the keyword argument *exc_info* with a true value, e.g.

logger.debug(“Houston, we have a %s”, “thorny problem”, *exc_info*=1)

error(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument `exc_info` with a true value, e.g.

`logger.error("Houston, we have a %s", "major problem", exc_info=1)`

exception(*msg, *args, exc_info=True, **kwargs*)

Convenience method for logging an ERROR with exception information.

info(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument `exc_info` with a true value, e.g.

`logger.info("Houston, we have a %s", "interesting problem", exc_info=1)`

warning(*msg, *args, **kwargs*)

Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument `exc_info` with a true value, e.g.

`logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)`

11.3 cubicweb.cwvreg

Cubicweb registries

class `cubicweb.cwvreg.CWRegistryStore`(*config, initlog=True*)

Bases: `logilab.common.registry.RegistryStore`

Central registry for the cubicweb instance, extending the generic `RegistryStore` with some cubicweb specific stuff.

This is one of the central object in cubicweb instance, coupling dynamically loaded objects with the schema and the configuration objects.

It specializes the `RegistryStore` by adding some convenience methods to access to stored objects. Currently we have the following registries of objects known by the web instance (library may use some others additional registries):

- 'etypes', entity type classes
- 'views', views and templates (e.g. layout views)
- 'components', non contextual components, like magic search, url evaluators
- 'ctxcomponents', contextual components like boxes and dynamic section
- 'actions', contextual actions, eg links to display in predefined places in the ui
- 'forms', describing logic of HTML form
- 'formrenderers', rendering forms to html
- 'controllers', primary objects to handle request publishing, directly plugged into the application

```
REGISTRY_FACTORY: Dict[Union[None, str], type] = {None: <class
'cubicweb.cwvreg.CWRegistry'>, 'etypes': <class 'cubicweb.cwvreg.ETypeRegistry'>,
'views': <class 'cubicweb.cwvreg.ViewsRegistry'>, 'actions': <class
'cubicweb.cwvreg.ActionsRegistry'>, 'ctxcomponents': <class
'cubicweb.cwvreg.CtxComponentsRegistry'>, 'uicfg': <class
'cubicweb.cwvreg.InstancesRegistry'>}
```

compute_var_types(*req, rqlst, args*)

init_properties(*propvalues*)

init the property values registry using the given set of couple (key, value)

initialization_completed()

cw specific code once vreg initialization is completed:

- remove objects requiring a missing appobject, unless `config.cleanup_unused_appobjects` is false
- init rtags

is_reload_needed(*modnames*)

overriden to handle modules names instead of directories

items() → a set-like object providing a view on D's items

iteritems()

itervalues()

load_file(*filepath, modname*)

load registrable objects (if any) from a python file

parse(*req, rql, args=None*)

property_info(*key*)

return dictionary containing description associated to the given property key (including type, default value, help and a site wide boolean)

property_value(*key*)

register(*obj, *args, **kwargs*)

register *obj* application object into *registryname* or *obj.__registry__* if not specified, with identifier *oid* or *obj.__regid__* if not specified.

If *clear* is true, all objects with the same identifier will be previously unregistered.

register_all(*objects, modname, butclasses=()*)

register registrable objects into *objects*.

Registrable objects are properly configured subclasses of `RegistrableObject`. Objects which are not defined in the module *modname* or which are in *butclasses* won't be registered.

Typical usage is:

```
store.register_all(globals().values(), __name__,
→(ClassIWantToRegisterExplicitly,))
```

So you get partially automatic registration, keeping manual registration for some object (to use `register_and_replace()` for instance).

register_and_replace(*obj, replaced*)

register *obj* object into *registryname* or *obj.__registries__* if not specified. If found, the *replaced* object will be unregistered first (else a warning will be issued as it is generally unexpected).

register_property(*key, type, help, default=None, vocabulary=None, sitewide=False*)

register a given property

reload(*modnames*, *force_reload=True*)

modification detected, reset and reload the vreg

reload_if_needed()

reset()

clear all registries managed by this store

property **rqlhelper**

set_schema(*schema*)

set instance's schema and load application objects

setdefault(*regid*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

typed_value(*key*, *value*)

value is a unicode string, return it correctly typed. Let potential type error propagates.

update_schema(*schema*)

update .schema attribute on registered objects, necessary for some tests

user_property_keys(*withsitewide=False*)

values() → an object providing a view on D's values

class `cubicweb.cwvreg.CWRegistry`(*vreg*)

Bases: `logilab.common.registry.Registry`

Parameters **vreg** – the `CWRegistryStore` managing this registry.

poss_visible_objects(**args*, ***kwargs*)

return an ordered list of possible app objects in a given registry, supposing they support the 'visible' and 'order' properties (as most visualizable objects)

property **schema**

The `cubicweb.schema.CubicWebSchema`

select(*_Registry__oid*, **args*, ***kwargs*)

return the most specific object among those with the given oid according to the given context.

raise `ObjectNotFound` if there are no object with id *oid* in this registry

raise `NoSelectableObject` if no object can be selected

class `cubicweb.cwvreg.InstancesRegistry`(*vreg*)

Bases: `cubicweb.cwvreg.CWRegistry`

Parameters **vreg** – the `CWRegistryStore` managing this registry.

selected(*winner*, *args*, *kwargs*)

overriden to avoid the default 'instanciation' behaviour, ie *winner*(**args*, ***kwargs*)

class `cubicweb.cwvreg.ETypeRegistry`(*vreg*)

Bases: `cubicweb.cwvreg.CWRegistry`

Parameters **vreg** – the `CWRegistryStore` managing this registry.

etype_class(**kwargs)

return an entity class for the given entity type.

Try to find out a specific class for this kind of entity or default to a dump of the nearest parent class (in yams inheritance) registered.

Fall back to 'Any' if not yams parent class found.

fetch_attrs(targettypes)

return intersection of fetch_attrs of each entity type in *targettypes*

initialization_completed()

on registration completed, clear etype_class internal cache

register(obj, **kwargs)

base method to add an object in the registry

class cubicweb.cwvreg.**ViewsRegistry**(vreg)

Bases: *cubicweb.cwvreg.CWRegistry*

Parameters *vreg* – the *CWRegistryStore* managing this registry.

main_template(req, oid='main-template', rset=None, **kwargs)

display query by calling the given template (default to main), and returning the output as a string instead of requiring the [w]rite method as argument

possible_views(req, rset=None, **kwargs)

return an iterator on possible views for this result set

views returned are classes, not instances

class cubicweb.cwvreg.**ActionsRegistry**(vreg)

Bases: *cubicweb.cwvreg.CWRegistry*

Parameters *vreg* – the *CWRegistryStore* managing this registry.

poss_visible_objects(*args, **kwargs)

return an ordered list of possible actions

class cubicweb.cwvreg.**CtxComponentsRegistry**(vreg)

Bases: *cubicweb.cwvreg.CWRegistry*

Parameters *vreg* – the *CWRegistryStore* managing this registry.

poss_visible_objects(*args, **kwargs)

return an ordered list of possible components

11.4 logilab.common.registry

This module provides bases for predicates dispatching (the pattern in use here is similar to what's referred as multi-dispatch or predicate-dispatch in the literature, though a bit different since the idea is to select across different implementation 'e.g. classes), not to dispatch a message to a function or method. It contains the following classes:

- *RegistryStore*, the top level object which loads implementation objects and stores them into registries. You'll usually use it to access registries and their contained objects;
- *Registry*, the base class which contains objects semantically grouped (for instance, sharing a same API, hence the 'implementation' name). You'll use it to select the proper implementation according to a context. Notice you may use registries on their own without using the store.

Note: implementation objects are usually designed to be accessed through the registry and not by direct instantiation, besides to use it as base classe.

The selection procedure is delegated to a selector, which is responsible for scoring the object according to some context. At the end of the selection, if an implementation has been found, an instance of this class is returned. A selector is built from one or more predicates combined together using AND, OR, NOT operators (actually `&`, `|` and `~`). You'll thus find some base classes to build predicates:

- `Predicate`, the abstract base predicate class
- `AndPredicate`, `OrPredicate`, `NotPredicate`, which you shouldn't have to use directly. You'll use `&`, `|` and `~` operators between predicates directly
- `objectify_predicate()`

You'll eventually find one concrete predicate: `yes`

class `logilab.common.registry.RegistryStore(debugmode: bool = False)`

This class is responsible for loading objects and storing them in their registry which is created on the fly as needed.

It handles dynamic registration of objects and provides a convenient api to access them. To be recognized as an object that should be stored into one of the store's registry (`Registry`), an object must provide the following attributes, used control how they interact with the registry:

`__registries__` list of registry names (string like 'views', 'templates'...) into which the object should be registered

`__regid__` object identifier in the registry (string like 'main', 'primary', 'folder_box')

`__select__` the object predicate selectors

Moreover, the `__abstract__` attribute may be set to `True` to indicate that an object is abstract and should not be registered (such inherited attributes not considered).

Note: When using the store to load objects dynamically, you *always* have to use `super()` to get the methods and attributes of the superclasses, and not use the class identifier. If not, you'll get into trouble at reload time.

For example, instead of writing:

```
class Thing(Parent):
    __regid__ = 'athing'
    __select__ = yes()

    def f(self, arg1):
        Parent.f(self, arg1)
```

You must write:

```
class Thing(Parent):
    __regid__ = 'athing'
    __select__ = yes()

    def f(self, arg1):
        super(Thing, self).f(arg1)
```

Dynamic loading is triggered by calling the `register_modnames()` method, given a list of modules names to inspect.

register_modnames(*modnames: List[str]*) → None

register all objects found in <modnames>

For each module, by default, all compatible objects are registered automatically. However if some objects come as replacement of other objects, or have to be included only if some condition is met, you'll have to define a `registration_callback(vreg)` function in the module and explicitly register **all objects** in this module, using the api defined below.

register_all(*objects: Iterable, modname: str, butclasses: Sequence = ()*) → None

register registrable objects into *objects*.

Registrable objects are properly configured subclasses of `RegistrableObject`. Objects which are not defined in the module *modname* or which are in *butclasses* won't be registered.

Typical usage is:

```
store.register_all(globals().values(), __name__,
→ (ClassIWantToRegisterExplicitly,))
```

So you get partially automatic registration, keeping manual registration for some object (to use `register_and_replace()` for instance).

register_and_replace(*obj, replaced, registryname=None*)

register *obj* object into *registryname* or *obj.__registries__* if not specified. If found, the *replaced* object will be unregistered first (else a warning will be issued as it is generally unexpected).

register(*obj: Any, registryname: Optional[Any] = None, oid: Optional[Any] = None, clear: bool = False*) → None

register *obj* implementation into *registryname* or *obj.__registries__* if not specified, with identifier *oid* or *obj.__regid__* if not specified.

If *clear* is true, all objects with the same identifier will be previously unregistered.

unregister(*obj, registryname=None*)

unregister *obj* object from the registry *registryname* or *obj.__registries__* if not specified.

Note: Once the function `registration_callback(vreg)` is implemented in a module, all the objects from this module have to be explicitly registered as it disables the automatic object registration.

Examples:

```
def registration_callback(store):
    # register everything in the module except BabarClass
    store.register_all(globals().values(), __name__, (BabarClass,))

    # conditionally register BabarClass
    if 'babar_relation' in store.schema:
        store.register(BabarClass)
```

In this example, we register all application object classes defined in the module except *BabarClass*. This class is then registered only if the 'babar_relation' relation type is defined in the instance schema.

```
def registration_callback(store):
    store.register(Elephant)
    # replace Babar by Celeste
    store.register_and_replace(Celeste, Babar)
```

In this example, we explicitly register classes one by one:

- the *Elephant* class
- the *Celeste* to replace *Babar*

If at some point we register a new appobject class in this module, it won't be registered at all without modification to the *registration_callback* implementation. The first example will register it though, thanks to the call to the *register_all* method.

The *REGISTRY_FACTORY* class dictionary allows to specify which class should be instantiated for a given registry name. The class associated to *None* key will be the class used when there is no specific class for a name.

class logilab.common.registry.**Registry**(*debugmode: bool*)

The registry store a set of implementations associated to identifier:

- to each identifier are associated a list of implementations
- to select an implementation of a given identifier, you should use one of the *select()* or *select_or_none()* method
- to select a list of implementations for a context, you should use the *possible_objects()* method
- dictionary like access to an identifier will return the bare list of implementations for this identifier.

To be usable in a registry, the only requirement is to have a *__select__* attribute.

At the end of the registration process, the *__registered__()* method is called on each registered object which have them, given the registry in which it's registered as argument.

Registration methods:

register(*obj: Any, oid: Optional[Any] = None, clear: bool = False*) → None
base method to add an object in the registry

unregister(*obj*)
remove object <obj> from this registry

Selection methods:

select(*_Registry__oid, *args, **kwargs*)
return the most specific object among those with the given oid according to the given context.
raise *ObjectNotFound* if there are no object with id *oid* in this registry
raise *NoSelectableObject* if no object can be selected

select_or_none(*_Registry__oid, *args, **kwargs*)
return the most specific object among those with the given oid according to the given context, or None if no object applies.

possible_objects(**args, **kwargs*)
return an iterator on possible objects in this registry for the given context

object_by_id(oid, *args, **kwargs)

return object with the *oid* identifier. Only one object is expected to be found.

raise *ObjectNotFound* if there are no object with id *oid* in this registry

raise *AssertionError* if there is more than one object there

11.4.1 Predicates

class logilab.common.registry.**Predicate**

base class for selector classes providing implementation for operators &, | and ~

This class is only here to give access to binary operators, the selector logic itself should be implemented in the `__call__()` method. Notice it should usually accept any arbitrary arguments (the context), though that may vary depending on your usage of the registry.

a selector is called to help choosing the correct object for a particular context by returning a score (*int*) telling how well the implementation given as first argument fit to the given context.

0 score means that the class doesn't apply.

logilab.common.registry.**objectify_predicate**(selector_func: Callable) → Any

Most of the time, a simple score function is enough to build a selector. The *objectify_predicate()* decorator turn it into a proper selector class:

```
@objectify_predicate
def one(cls, req, rset=None, **kwargs):
    return 1

class MyView(View):
    __select__ = View.__select__ & one()
```

class logilab.common.registry.**yes**(score: float = 0.5)

Return the score given as parameter, with a default score of 0.5 so any other selector take precedence.

Usually used for objects which can be selected whatever the context, or also sometimes to add arbitrary points to a score.

Take care, *yes(0)* could be named 'no'...

class logilab.common.registry.**AndPredicate**(*selectors: Any)

and-chained selectors

class logilab.common.registry.**OrPredicate**(*selectors: Any)

or-chained selectors

class logilab.common.registry.**NotPredicate**(selector)

negation selector

11.4.2 Debugging

class logilab.common.registry.traced_selection(*traced='all'*)

Typical usage is :

```
>>> from logilab.common.registry import traced_selection
>>> with traced_selection():
...     # some code in which you want to debug selectors
...     # for all objects
```

This will yield lines like this in the logs:

```
selector one_line_rset returned 0 for <class 'elephant.Babar'>
```

You can also give to *traced_selection* the identifiers of objects on which you want to debug selection ('oid1' and 'oid2' in the example above).

```
>>> with traced_selection( ('regid1', 'regid2') ):
...     # some code in which you want to debug selectors
...     # for objects with __regid__ 'regid1' and 'regid2'
```

A potentially useful point to set up such a tracing function is the *logilab.common.registry.Registry.select* method body.

11.4.3 Exceptions

class logilab.common.registry.RegistryException

Base class for registry exception.

class logilab.common.registry.RegistryNotFound

Raised when an unknown registry is requested.

This is usually a programming/typo error.

class logilab.common.registry.ObjectNotFound

Raised when an unregistered object is requested.

This may be a programming/typo or a misconfiguration error.

class logilab.common.registry.NoSelectableObject(*args, kwargs, objects*)

Raised when no object is selectable for a given context.

11.5 cubicweb.dataimport

Package containing various utilities to import data into cubicweb.

11.5.1 Utilities

`cubicweb.dataimport.count_lines(*args, **kwargs)`

`cubicweb.dataimport.ucsvreader_pb(*args, **kwargs)`

`cubicweb.dataimport.ucsvreader(*args, **kwargs)`

`cubicweb.dataimport.callfunc_every(func, number, iterable)`

yield items of *iterable* one by one and call function *func* every *number* iterations. Always call function *func* at the end.

11.5.2 Object Stores

`cubicweb.dataimport.RQLObjectStore`

alias of `logilab.common.deprecation.callable_moved.<locals>.callnew`

`cubicweb.dataimport.NoHookRQLObjectStore`

alias of `logilab.common.deprecation.callable_moved.<locals>.callnew`

11.6 cubicweb.predicates

Predicate classes

class `cubicweb.appobject.yes(score: float = 0.5)`

Return the score given as parameter, with a default score of 0.5 so any other selector take precedence.

Usually used for objects which can be selected whatever the context, or also sometimes to add arbitrary points to a score.

Take care, *yes(0)* could be named 'no'...

class `cubicweb.predicates.match_kwargs(*expected, **kwargs)`

Return non-zero score if parameter names specified as initializer arguments are specified in the input context.

Return a score corresponding to the number of expected parameters.

When multiple parameters are expected, all of them should be found in the input context unless *mode* keyword argument is given to 'any', in which case a single matching parameter is enough.

class `cubicweb.predicates.appobject_selectable(registry, *regids)`

Return 1 if another appobject is selectable using the same input context.

Initializer arguments:

- *registry*, a registry name
- *regids*, object identifiers in this registry, one of them should be selectable.

class `cubicweb.predicates.adaptable(*regids)`

Return 1 if another appobject is selectable using the same input context.

Initializer arguments:

- *regids*, adapter identifiers (e.g. interface names) to which the context (usually entities) should be adaptable. One of them should be selectable when multiple identifiers are given.

class cubicweb.predicates.configuration_values(*key, values*)

Return 1 if the instance has an option set to a given value(s) in its configuration file.

class cubicweb.predicates.none_rset

Return 1 if the result set is None (eg usually not specified).

class cubicweb.predicates.any_rset

Return 1 for any result set, whatever the number of rows in it, even 0.

class cubicweb.predicates.nonempty_rset

Return 1 for result set containing one ore more rows.

class cubicweb.predicates.empty_rset

Return 1 for result set which doesn't contain any row.

class cubicweb.predicates.one_line_rset

Return 1 if the result set is of size 1, or greater but a specific row in the result set is specified ('row' argument).

class cubicweb.predicates.multi_lines_rset(*expected=None, operator=<built-in function eq>*)

Return 1 if the operator expression matches between *num* elements in the result set and the *expected* value if defined.

By default, multi_lines_rset(expected) matches equality expression: *nb* row(s) in result set equals to expected value

But, you can perform richer comparisons by overriding default operator: multi_lines_rset(expected, operator.gt)

If *expected* is None, return 1 if the result set contains *at least* two rows. If rset is None, return 0.

class cubicweb.predicates.multi_columns_rset(*expected=None, operator=<built-in function eq>*)

If *nb* is specified, return 1 if the result set has exactly *nb* column per row. Else (*nb* is None), return 1 if the result set contains *at least* two columns per row. Return 0 for empty result set.

class cubicweb.predicates.paginated_rset(*nbpages=1*)

Return 1 or more for result set with more rows than one or more page size. You can specify expected number of pages to the initializer (default to one), and you'll get that number of pages as score if the result set is big enough.

Page size is searched in (respecting order): * a *page_size* argument * a *page_size* form parameters * the *navigation.page-size* property (see [Configuring persistent properties](#))

class cubicweb.predicates.sorted_rset

Return 1 for sorted result set (e.g. from an RQL query containing an ORDERBY clause), with exception that it will return 0 if the rset is 'ORDERBY FTIRANK(VAR)' (eg sorted by rank value of the has_text index).

class cubicweb.predicates.one_etype_rset

Return 1 if the result set contains entities which are all of the same type in the column specified by the *col* argument of the input context, or in column 0.

class cubicweb.predicates.multi_etypes_rset(*expected=None, operator=<built-in function eq>*)

If *nb* is specified, return 1 if the result set contains *nb* different types of entities in the column specified by the *col* argument of the input context, or in column 0. If *nb* is None, return 1 if the result set contains *at least* two different types of entities.

class cubicweb.predicates.non_final_entity(*accept_none=True, mode='all'*)

Return 1 for entity of a non final entity type(s). Remember, "final" entity types are String, Int, etc... This is equivalent to *is_instance('Any')* but more optimized.

See EClassPredicate documentation for entity class lookup / score rules according to the input context.

class cubicweb.predicates.**is_instance**(*expected_etypes, **kwargs)

Return non-zero score for entity that is an instance of the one of given type(s). If multiple arguments are given, matching one of them is enough.

Entity types should be given as string, the corresponding class will be fetched from the registry at selection time.

See `EClassPredicate` documentation for entity class lookup / score rules according to the input context.

Note: the score will reflect class proximity so the most specific object will be selected.

class cubicweb.predicates.**score_entity**(scorefunc, mode='all')

Return score according to an arbitrary function given as argument which will be called with input content entity as argument.

This is a very useful predicate that will usually interest you since it allows a lot of things without having to write a specific predicate.

The function can return arbitrary value which will be casted to an integer value at the end.

See `EntityPredicate` documentation for entity lookup / score rules according to the input context.

class cubicweb.predicates.**rql_condition**(expression, mode='all', user_condition=False)

Return non-zero score if arbitrary rql specified in *expression* initializer argument return some results for entity found in the input context. Returned score is the number of items returned by the rql condition.

expression is expected to be a string containing an rql expression, which must use 'X' variable to represent the context entity and may use 'U' to represent the request's user.

Warning: If simply testing value of some attribute/relation of context entity (X), you should rather use the `score_entity` predicate which will benefit from the ORM's request entities cache.

See `EntityPredicate` documentation for entity lookup / score rules according to the input context.

class cubicweb.predicates.**relation_possible**(rtype, role='subject', target_etype=None, action='read', strict=False, **kwargs)

Return 1 for entity that supports the relation, provided that the request's user may do some *action* on it (see below).

The relation is specified by the following initializer arguments:

- *rtype*, the name of the relation
- *role*, the role of the entity in the relation, either 'subject' or 'object', default to 'subject'
- *target_etype*, optional name of an entity type that should be supported at the other end of the relation
- *action*, a relation schema action (e.g. one of 'read', 'add', 'delete', default to 'read') which must be granted to the user, else a 0 score will be returned. Give None if you don't want any permission checking.
- *strict*, boolean (default to False) telling what to do when the user has not globally the permission for the action (eg the action is not granted to one of the user's groups)
 - when *strict* is False, if there are some local role defined for this action (e.g. using rql expressions), then the permission will be considered as granted
 - when *strict* is True, then the permission will be actually checked for each entity

Setting *strict* to True impacts performance for large result set since you'll then get the `EntityPredicate` behaviour while otherwise you get the `EClassPredicate`'s one. See those classes documentation for entity lookup / score rules according to the input context.

class `cubicweb.predicates.partial_relation_possible`(*action='read', **kwargs*)

Same as `:class:~`cubicweb.predicates.relation_possible``, but will look for attributes of the selected class to get information which is otherwise expected by the initializer, except for *action* and *strict* which are kept as initializer arguments.

This is useful to predefine predicate of an abstract class designed to be customized.

class `cubicweb.predicates.has_related_entities`(*rtype, role='subject', target_etype=None, **kwargs*)

Return 1 if entity support the specified relation and has some linked entities by this relation , optionally filtered according to the specified target type.

The relation is specified by the following initializer arguments:

- *rtype*, the name of the relation
- *role*, the role of the entity in the relation, either 'subject' or 'object', default to 'subject'.
- *target_etype*, optional name of an entity type that should be found at the other end of the relation

See `EntityPredicate` documentation for entity lookup / score rules according to the input context.

class `cubicweb.predicates.partial_has_related_entities`(***kwargs*)

Same as `:class:~`cubicweb.predicates.has_related_entities``, but will look for attributes of the selected class to get information which is otherwise expected by the initializer.

This is useful to predefine predicate of an abstract class designed to be customized.

class `cubicweb.predicates.has_permission`(*action*)

Return non-zero score if request's user has the permission to do the requested action on the entity. *action* is an entity schema action (eg one of 'read', 'add', 'delete', 'update').

Here are entity lookup / scoring rules:

- if *entity* is specified, check permission is granted for this entity
- elif *row* is specified, check permission is granted for the entity found in the specified cell
- else check permission is granted for each entity found in the column specified specified by the *col* argument or in column 0

class `cubicweb.predicates.has_add_permission`(*etype=None, **kwargs*)

Return 1 if request's user has the add permission on entity type specified in the *etype* initializer argument, or according to entity found in the input content if not specified.

It also check that then entity type is not a strict subobject (e.g. may only be used as a composed of another entity).

See `EClassPredicate` documentation for entity class lookup / score rules according to the input context when *etype* is not specified.

class `cubicweb.predicates.has_mimetype`(*mimetype, mode='all'*)

Return 1 if the entity adapt to `IDownloadable` and has the given MIME type.

You can give 'image/' to match any image for instance, or 'image/png' to match only PNG images.

class `cubicweb.predicates.is_in_state`(**expected*)

Return 1 if entity is in one of the states given as argument list

You should use this instead of your own `score_entity` predicate to avoid some gotchas:

- possible views gives a fake entity with no state

- you must use the latest `tr` info thru the workflow adapter for repository side checking of the current state

In debug mode, this predicate can raise `ValueError` for unknown states names (only checked on entities without a custom workflow)

Return type `int`

`cubicweb.predicates.on_fire_transition(etype, tr_names)`

Return 1 when entity of the type *etype* is going through transition of a name included in *tr_names*.

You should use this predicate on ‘after_add_entity’ hook, since it’s actually looking for addition of *TrInfo* entities. Hence in the hook, *self.entity* will reference the matching *TrInfo* entity, allowing to get all the transition details (including the entity to which is applied the transition but also its original state, transition, destination state, user...).

See `cubicweb.entities.wfobjs.TrInfo` for more information.

class `cubicweb.predicates.match_user_groups(*expected, **kwargs)`

Return a non-zero score if request’s user is in at least one of the groups given as initializer argument. Returned score is the number of groups in which the user is.

If the special ‘owners’ group is given and *rset* is specified in the input context:

- if *row* is specified check the entity at the given *row/col* (default to 0) is owned by the user
- else check all entities in *col* (default to 0) are owned by the user

class `cubicweb.predicates.no_cnx`

Return 1 if the web session has no connection set. This occurs when anonymous access is not allowed and user isn’t authenticated.

class `cubicweb.predicates.anonymous_user`

Return 1 if the user is not authenticated (i.e. is the anonymous user).

class `cubicweb.predicates.authenticated_user`

Return 1 if the user is authenticated (i.e. not the anonymous user).

class `cubicweb.predicates.match_form_params(*expected, **kwargs)`

Return non-zero score if parameter names specified as initializer arguments are specified in request’s form parameters.

Return a score corresponding to the number of expected parameters.

When multiple parameters are expected, all of them should be found in the input context unless *mode* keyword argument is given to ‘any’, in which case a single matching parameter is enough.

override default `__init__` to allow either named or positional parameters.

class `cubicweb.predicates.match_search_state(*expected, **kwargs)`

Return 1 if the current request search state is in one of the expected states given to the initializer.

Known search states are either ‘normal’ or ‘linksearch’ (eg searching for an object to create a relation with another).

This predicate is usually used by action that want to appears or not according to the ui search state.

class `cubicweb.predicates.match_context_prop`

Return 1 if:

- no *context* is specified in input context (take care to confusion, here *context* refers to a string given as an argument to the input context...)
- specified *context* is matching the context property value for the appobject using this predicate

- the appobject's context property value is None

This predicate is usually used by contextual components that want to appears in a configurable place.

```
class cubicweb.predicates.match_context(*expected, **kwargs)
```

```
class cubicweb.predicates.match_view(*expected, **kwargs)
```

Return 1 if a view is specified an as its registry id is in one of the expected view id given to the initializer.

```
class cubicweb.predicates.primary_view
```

Return 1 if:

- *no view is specified* in the input context
- a view is specified and its *.is_primary()* method return True

This predicate is usually used by contextual components that only want to appears for the primary view of an entity.

```
class cubicweb.predicates.contextual
```

Return 1 if view's contextual property is true

```
class cubicweb.predicates.specified_etype_implements(*expected_etypes, **kwargs)
```

Return non-zero score if the entity type specified by an 'etype' key searched in (by priority) input context kwargs and request form parameters match a known entity type (case insensitvly), and it's associated entity class is of one of the type(s) given to the initializer. If multiple arguments are given, matching one of them is enough.

Note: as with *is_instance*, entity types should be given as string and the score will reflect class proximity so the most specific object will be selected.

This predicate is usually used by views holding entity creation forms (since we've no result set to work on).

```
class cubicweb.predicates.attribute_edited(attribute, mode='all')
```

Scores if the specified attribute has been edited This is useful for selection of forms by the edit controller.

The initial use case is on a form, in conjunction with *match_transition*, which will not score at edit time:

```
is_instance('Version') & (match_transition('ready') |
                           attribute_edited('publication_date'))
```

```
class cubicweb.predicates.match_transition(*expected, **kwargs)
```

Return 1 if *transition* argument is found in the input context which has a *.name* attribute matching one of the expected names given to the initializer.

This predicate is expected to be used to customise the status change form in the web ui.

```
class cubicweb.predicates.match_exception(*expected)
```

Return 1 if exception given as *exc* in the input context is an instance of one of the class given on instantiation of this predicate.

```
class cubicweb.predicates.debug_mode
```

Return 1 if running in debug mode.

11.7 cubicweb.pyramid

Pyramid interface to CubicWeb

`cubicweb.pyramid.wsgi_application_from_cwconfig(cwconfig, debugtoolbar=False)`

Build a WSGI application from a cubicweb configuration

Parameters

- **cwconfig** – A CubicWeb configuration
- **profile** – Enable profiling. See *Profiling and performance*.
- **profile_output** – Profiling output filename. See *Profiling and performance*.
- **profile_dump_every** – Profiling number of requests before dumping the stats. See *Profiling and performance*.
- **debugtoolbar** – Activate pyramid debugtoolbar when True.

Returns A fully operationnal WSGI application

`cubicweb.pyramid.wsgi_application(instance_name=None, debug=None)`

Build a WSGI application from a cubicweb instance name

Parameters

- **instance_name** – Name of the cubicweb instance (optional). If not provided, `CW_INSTANCE` must exists.
- **debug** – Enable/disable the debug mode. If defined to True or False, overrides `CW_DEBUG`.

The following environment variables are used if they exist:

CW_INSTANCE

A CubicWeb instance name.

CW_DEBUG

If defined, the debugmode is enabled.

The function can be used as an entry-point for third-party wsgi containers. Below is a sample uswgi configuration file:

```
[uwsgi]
http = 127.0.1.1:8080
env = CW_INSTANCE=myinstance
env = CW_DEBUG=1
module = cubicweb.pyramid:wsgi_application()
virtualenv = /home/user/.virtualenvs/myvirtualenv
processes = 1
threads = 8
stats = 127.0.0.1:9191
plugins = http,python
```

11.7.1 cubicweb.pyramid.auth

CubicWeb AuthTkt authentication policy

When using the *cubicweb.pyramid.auth* module, which is the default in most cases, you may have to configure the behaviour of these authentication policies using standard's Pyramid configuration. You may want to configure in your pyramid configuration file:

Session Authentication This is a [AuthTktAuthenticationPolicy](#) so you may overwrite default configuration values by adding configuration entries using the prefix `cubicweb.auth.auth_tkt.session`. Default values are:

```
cubicweb.auth.auth_tkt.session.hashalg = sha512
cubicweb.auth.auth_tkt.session.cookie_name = auth_tkt
cubicweb.auth.auth_tkt.session.timeout = 1200
cubicweb.auth.auth_tkt.session.reissue_time = 120
cubicweb.auth.auth_tkt.session.http_only = True
cubicweb.auth.auth_tkt.session.secure = True
```

Persistent Authentication This is also a [AuthTktAuthenticationPolicy](#). It is used when persistent sessions are activated (typically when using the [cubicweb-rememberme](#) cube). You may overwrite default configuration values by adding configuration entries using the prefix `cubicweb.auth.auth_tkt.persistent`. Default values are:

```
cubicweb.auth.auth_tkt.persistent.hashalg = sha512
cubicweb.auth.auth_tkt.persistent.cookie_name = pauth_tkt
cubicweb.auth.auth_tkt.persistent.max_age = 3600*24*30
cubicweb.auth.auth_tkt.persistent.reissue_time = 3600*24
cubicweb.auth.auth_tkt.persistent.http_only = True
cubicweb.auth.auth_tkt.persistent.secure = True
```

Warning: Legacy timeout values from the instance's `all-in-one.conf` are **not** used at all (`http-session-time` and `cleanup-session-time`)

Secrets

There are a number of secrets to configure in `pyramid.ini`. They should be different one from each other, as explained in [`Pyramid's documentation`_](#).

For the record, regarding authentication:

cubicweb.auth.auth_tkt.session.secret This secret is used to encrypt the authentication cookie.

cubicweb.auth.auth_tkt.persistent.secret This secret is used to encrypt the persistent authentication cookie.

`cubicweb.pyramid.auth.includeme`(*config*)

Activate the CubicWeb AuthTkt authentication policy.

Usually called via `config.include('cubicweb.pyramid.auth')`.

See also [cubicweb.pyramid.defaults](#)

class cubicweb.pyramid.auth.UpdateLoginTimeAuthenticationPolicy

Bases: object

An authentication policy that update the user last_login_time.

The update is done in the ‘remember’ method, which is called by the login views login,

Usually used via *includeme()*.

11.7.2 cubicweb.pyramid.bwcompat

Backward compatibility layer for CubicWeb to run as a Pyramid application.

cubicweb.pyramid.bwcompat.includeme(*config*)

Set up a tween app that will handle the request if the main application raises a HTTPNotFound exception.

This is to keep legacy compatibility for cubes that makes use of the cubicweb urlresolvers.

It provides, for now, support for cubicweb controllers, but this feature will be reimplemented separately in a less compatible way.

It is automatically included by the configuration system, but can be disabled in the *Pyramid Settings file*:

```
cubicweb.bwcompat = no
```

class cubicweb.pyramid.bwcompat.PyramidSessionHandler(*appli*)

A CW Session handler that rely on the pyramid API to fetch the needed informations.

It implements the cubicweb.web.application.CookieSessionHandler API.

class cubicweb.pyramid.bwcompat.CubicWebPyramidHandler(*appli, cubicweb_config*)

A Pyramid request handler that rely on a cubicweb instance to do the whole job

Parameters *appli* – A CubicWeb ‘Application’ object.

__call__(*request*)

Handler that mimics what CubicWebPublisher.main_handle_request and CubicWebPublisher.core_handle do

class cubicweb.pyramid.bwcompat.TweenHandler(*handler, registry*)

A Pyramid tween handler that submit unhandled requests to a Cubicweb handler.

The CubicWeb handler to use is expected to be in the pyramid registry, at key 'cubicweb.handler'.

11.7.3 cubicweb.pyramid.core

Binding of CubicWeb connection to Pyramid request.

cubicweb.pyramid.core.includeme(*config*)

Enables the core features of Pyramid CubicWeb.

Automatically called by the ‘pyramid’ command, or via *config.include('cubicweb.pyramid.code')*. In the later case, the following registry entries must be defined first:

‘**cubicweb.config**’ A cubicweb ‘config’ instance.

‘**cubicweb.repository**’ The corresponding cubicweb repository.

‘**cubicweb.registry**’ The vreg.

`cubicweb.pyramid.core.cw_to_pyramid(request)`

Context manager to wrap a call to the cubicweb API.

All CW exceptions will be transformed into their pyramid equivalent. When needed, some CW response bits may be converted too (mainly headers)

`cubicweb.pyramid.core.render_view(request, vid, **kwargs)`

Helper function to render a CubicWeb view.

Parameters

- **request** – A pyramid request
- **vid** – A CubicWeb view id
- **kwargs** – Keyword arguments to select and instantiate the view

Returns The rendered view content

`cubicweb.pyramid.core.repo_connect(request, repo, eid)`

A lightweight version of `cubicweb.server.repository.Repository.connect()` that does not keep track of opened sessions, removing the need of closing them

`cubicweb.pyramid.core.get_principals(login, request)`

Returns the group names of the authenticated user.

This function is meant to be used as an authentication policy callback.

It also pre-open the cubicweb session and put it in `request.cw_cached_session` for later usage by `_cw_session()`.

Note: If the default authentication policy is not used, make sure this function gets called by the active authentication policy.

Parameters

- **login** – A cubicweb user eid
- **request** – A pyramid request

Returns A list of group names

`class cubicweb.pyramid.core.CubicWebPyramidRequest(request)`

Bases: `cubicweb.web.request.ConnectionCubicWebRequestBase`

A CubicWeb request that only wraps a pyramid request.

Parameters **request** – A pyramid request

http_method()

returns 'POST', 'GET', 'HEAD', etc.

property message

Returns a '
' joined list of the cubicweb current message and the default pyramid flash queue messages.

relative_path(includeparams=True)

return the normalized path of the request (ie at least relative to the instance's root, but some other normalization may be needed so that the returned path may be used to compare to generated urls

Parameters **includeparams** – boolean indicating if GET form parameters should be kept in the path

setup_params(*params*)

WARNING: we're intentionally leaving INTERNAL_FIELD_VALUE here
subclasses should overrides to

`cubicweb.pyramid.core._cw_session(request)`

Obtains a cw session from a pyramid request

Parameters **request** – A pyramid request

Returns type `cubicweb.server.session.Session`

Not meant for direct use, use `request.cw_session` instead.

`cubicweb.pyramid.core._cw_cnx(request)`

Obtains a cw session from a pyramid request

The connection will be committed or rolled-back in a request finish callback (this is temporary, we should make use of the transaction manager in a later version).

Not meant for direct use, use `request.cw_cnx` instead.

Parameters **request** – A pyramid request

Returns type `cubicweb.server.session.Connection`

`cubicweb.pyramid.core._cw_request(request)`

Obtains a CubicWeb request wrapper for the pyramid request.

Parameters **request** – A pyramid request

Returns A CubicWeb request

Returns type `CubicWebPyramidRequest`

Not meant for direct use, use `request.cw_request` instead.

11.7.4 cubicweb.pyramid.defaults

Defaults for a classical CubicWeb instance.

`cubicweb.pyramid.defaults.includeme(config)`

Enable the defaults that make the application behave like a classical CubicWeb instance.

The following modules get included:

- `cubicweb.pyramid.session`
- `cubicweb.pyramid.auth`
- `cubicweb.pyramid.login`

It is automatically included by the configuration system, unless the following entry is added to the *Pyramid Settings file*:

```
cubicweb.defaults = no
```

11.7.5 cubicweb.pyramid.login

Provide login views that reproduce a classical CubicWeb behavior

`cubicweb.pyramid.login.includeme(config)`

Create the 'login' route ('/login') and load this module views

Views

`cubicweb.pyramid.login.login_form(request)`

Default view for the 'login' route.

Display the 'login' CubicWeb view, which is should be a login form

`cubicweb.pyramid.login.login_password_login(request)`

Handle POST of `__login/``__password` on the 'login' route.

The authentication itself is delegated to the CubicWeb repository.

Request parameters:

Parameters

- `__login` – The user login (or email if `allow-email-login` is on).
- `__password` – The user password
- `__setauthcookie` – (optional) If defined and equal to '1', set the authentication cookie maxage to 1 week.

If not, the authentication cookie is a session cookie.

`cubicweb.pyramid.login.login_already_loggedin(request)`

'login' route view for Authenticated users.

Simply redirect the user to '/

11.7.6 cubicweb.pyramid.profile

11.7.7 cubicweb.pyramid.session

Web session when using pyramid

CubicWeb `CWSession` entity type so that sessions can be stored in the database, which allows to run a Cubicweb instance without having to set up a session storage (like redis or memcache) solution.

However, for production systems, it is greatly advised to use such a storage solution for the sessions.

The handling of the sessions is made by pyramid (see the [`pyramid's documentation on sessions`_](#) for more details).

For example, to set up a redis based session storage, you need the [`pyramid-redis-session`_](#) package, then you must configure pyramid to use this backend, by configuring the pyramid configuration file:

```
[main]
cubicweb.defaults = no # we do not want to load the default cw session handling

cubicweb.auth.authkt.session.secret = <secret1>
cubicweb.auth.authkt.persistent.secret = <secret2>
```

(continues on next page)

(continued from previous page)

```

cubicweb.auth.authkt.session.secure = yes
cubicweb.auth.authkt.persistent.secure = yes

redis.sessions.secret = <secret3>
redis.sessions.prefix = <my-app>:

redis.sessions.url = redis://localhost:6379/0

pyramid.includes =
    pyramid_redis_sessions
    cubicweb.pyramid.auth
    cubicweb.pyramid.login

```

Warning: If you want to be able to log in a CubicWeb application served by pyramid on a unsecured stream (typically when you start an instance in dev mode using a simple `cubicweb-ctl pyramid -D -linfo myinstance`), you **must** set `cubicweb.auth.authkt.session.secure` to no.

Secrets

There are a number of secrets to configure in `pyramid.ini`. They should be different one from each other, as explained in [`Pyramid's documentation`_](#).

For the record, regarding session handling:

cubicweb.session.secret This secret is used to encrypt the session's data ID (data themselves are stored in the backend, database or redis) when using the integrated (CWSession based) session data storage.

redis.session.secret This secret is used to encrypt the session's data ID (data themselves are stored in the backend, database or redis) when using redis as backend.

`cubicweb.pyramid.session.includeme`(*config*)

Activate the CubicWeb session factory.

Usually called via `config.include('cubicweb.pyramid.auth')`.

See also [cubicweb.pyramid.defaults](#)

`cubicweb.pyramid.session.CWSessionFactory`(*secret, cookie_name='session', max_age=None, path='/', domain=None, secure=False, httponly=True, set_on_exception=True, timeout=1200, reissue_time=120, hashalg='sha512', salt='pyramid.session.', serializer=None*)

A pyramid session factory that store session data in the CubicWeb database.

Storage is done with the 'CWSession' entity, which is provided by the 'pyramid' cube.

Warning: Although it provides a sane default behavior, this session storage has a serious overhead because it uses RQL to access the database.

Using pure SQL would improve a bit (it is roughly twice faster), but it is still pretty slow and thus not an immediate priority.

It is recommended to use faster session factory ([pyramid_redis_sessions](#) for example) if you need speed.

11.7.8 cubicweb.pyramid.url_redirection

Url redirection using pyramid

This module allow to define redirection rules used by pyramid before route selection.

Each rule has to be added with `add_rewriting_rule` method in a pyramid *includeme* function to be used.

Example of usage :

```
def includeme(config):
    config.add_rewriting_rule(r'<a regex>', callback_method)
```

The `callback_method` takes three arguments:

- a pyramid request ;
- the url that has to be matched by the rule ;
- a dictionary containing all the named subgroups of the regex match.

It must return a string corresponding to the new url.

`cubicweb.pyramid.url_redirection.includeme(config)`

Add `add_redirection_rule` pyramid directive and `url_redirection_tween` pyramid tween handler to the pyramid configuration.

`cubicweb.pyramid.url_redirection.add_redirection_rule(config, rule, callback_method)`

Declare a pyramid directive allowing to add a redirection rule. Each rewriting rule can then be added in the *includeme* at the end of the file by using `config.add_redirection_rule(rule, callback_method)`. :param config: pyramid configuration, used by pyramid. :param rule: a regex used to match uri. :param callback_method: a method which return the redirected uri.

`cubicweb.pyramid.url_redirection.url_redirection_tween_factory(handler, registry)`

A pyramid tween handler that browse each *redirection_rules* added with `add_redirection_rule` directive to find if the current path matches a rule.

11.8 cubicweb.req

Base class for request/session

class `cubicweb.req.RequestSessionBase(vreg)`

base class containing stuff shared by server session and web request

request/session is the main resources accessor, mainly through it's *vreg* attribute:

Attribute `vreg` the instance's registry

Attribute `vreg.schema` the instance's schema

Attribute `vreg.config` the instance's configuration

base_url()

Return the root url of the instance.

build_url(*args, **kwargs)

return an absolute URL using params dictionary key/values as URL parameters. Values are automatically URL quoted, and the publishing method to use may be specified or will be guessed.

raises `ValueError` if None is found in arguments

build_url_params(**kwargs)

return encoded params to incorporate them in a URL

build_url_path(method, kwargs)

return the “path” part of an URL

create_entity(etype, **kwargs)

add a new entity of the given type

Example (in a shell session):

```
>>> c = create_entity('Company', name=u'Logilab')
>>> create_entity('Person', firstname=u'John', surname=u'Doe',
...               works_for=c)
```

eid_rset(eid, etype=None)

return a result set for the given eid without doing actual query (we have the eid, we can suppose it exists and user has access to the entity)

empty_rset()

return a guaranteed empty result

ensure_ro_rql(rql)

raise an exception if the given rql is not a select query

entity_from_eid(eid, etype=None)

return an entity instance for the given eid. No query is done

etype_rset(etype, size=1)

return a fake result set for a particular entity type

exists(etype, **kwargs)

return if it exists at least one entity of the given type and attribute values.

```
>>> is_stallman_there = cnx.exists('CWUser', login=u"rms")
>>> # more optimized than
>>> is_stallman_there = bool(cnx.find('CWUser', login=u"rms"))
>>> # it did a full scan table :-/
```

find(etype, limit=None, **kwargs)

find entities of the given type and attribute values.

Parameters

- **etype** – the type of entities to return.
- **limit** – the max number of entities to return.

Returns A ResultSet

```
>>> users = find('CWGroup', name=u"users").one()
>>> groups = find('CWGroup').entities()
```

format_date(date, date_format=None, time=False)

return a string for a date time according to instance’s configuration

format_float(num)

return a string for floating point number according to instance’s configuration

format_time(*time*)

return a string for a time according to instance's configuration

parse_datetime(*value*, *etype*='Datetime')

get a datetime or time from a string (according to etype) Datetime formatted as Date are accepted

printable_value(*attrtype*, *value*, *props*=None, *displaytime*=True, *formatters*={'BigInt': <function print_int>, 'Boolean': <function print_boolean>, 'Bytes': <function print_bytes>, 'Date': <function print_date>, 'Datetime': <function print_datetime>, 'Decimal': <function print_float>, 'Float': <function print_float>, 'Int': <function print_int>, 'Interval': <function print_timedelta>, 'String': <function print_string>, 'TZDatetime': <function print_tzdatetime>, 'TZTime': <function print_tztime>, 'Time': <function print_time>})

return a displayable value (i.e. unicode string)

property_value(*key*)

return value of the property with the given key, giving priority to user specific value if any, else using site value

rebuild_url(*url*, ***newparams*)

return the given url with newparams inserted. If any new params is already specified in the url, it's overridden by the new value

newparams may only be mono-valued.

set_language(*lang*)

install i18n configuration for *lang* translation.

Raises `KeyError` if translation doesn't exist.

url_parse_qsl(*querystring*)

return a list of (key, val) found in the url quoted query string

url_quote(*value*, *safe*='')

urllib.quote is not unicode safe, use this method to do the necessary encoding / decoding. Also it's designed to quote each part of a url path and so the '/' character will be encoded as well.

url_unquote(*quoted*)

returns a unicode unquoted string

decoding is based on *self.encoding* which is the encoding used in *url_quote*

user_data(***kwargs*)

returns a dictionary with this user's information.

The keys are :

login The user login

name The user name, returned by *user.name()*

email The user principal email

view(*_RequestSessionBase__vid*, *rset*=None, *_RequestSessionBase__fallback_oid*=None, *_RequestSessionBase__registry*='views', *initargs*=None, *w*=None, ***kwargs*)

Select object with the given id (*__oid*) then render it. If the object isn't selectable, try to select fallback object if *__fallback_oid* is specified.

If specified *initargs* is expected to be a dictionary containing arguments that should be given to selection (hence to object's *__init__* as well), but not to *render()*. Other arbitrary keyword arguments will be given to selection *and* to *render()*, and so should be handled by object's *call* or *cell_call* method..

11.9 cubicweb.rset

The *ResultSet* class which is returned as result of an rql query

class cubicweb.rset.**ResultSet**(*results, rql, args=None, description=None, variables=None*)

A result set wraps a RQL query result. This object implements partially the list protocol to allow direct use as a list of result rows.

Parameters

- **rowcount** (*int*) – number of rows in the result
- **rows** (*list*) – list of rows of result
- **description** (*list*) – result’s description, using the same structure as the result itself
- **rql** (*str or unicode*) – the original RQL query string

all(*col=0*)

iter on entities with eid in the *col* column of the result set

column_types(***kwargs*)

return the list of different types in the column with the given *col*

Parameters **col** (*int*) – the index of the desired column

Return type *list*

Returns the different entities type found in the column

complete_entity(*row, col=0, skip_bytes=True*)

short cut to get an completed entity instance for a particular row (all instance’s attributes have been fetched)

description_struct(***kwargs*)

return a list describing sequence of results with the same description, e.g. : `[[0, 4, ('Bug',)] [[0, 4, ('Bug',), [5, 8, ('Story',)] [[0, 3, ('Project', 'Version',)]]`

entities(*col=0*)

iter on entities with eid in the *col* column of the result set

filtered_rset(*filtercb, col=0*)

filter the result set according to a given filtercb

Parameters

- **filtercb** (*callable(entity)*) – a callable which should take an entity as argument and return False if it should be skipped, else True
- **col** (*int*) – the column index

Return type *ResultSet*

first(*col=0*)

Retrieve the first entity from the query.

If the result set is empty, raises `NoResultError`.

Parameters **col** (*int*) – The column localising the entity in the unique row

Returns the partially initialized *Entity* instance

get_entity(kwargs)**

convenience method for query retrieving a single entity, returns a partially initialized Entity instance.

Warning: Due to the cache wrapping this function, you should NEVER give row as a named parameter (i.e. `rset.get_entity(0, 1)` is OK but `rset.get_entity(row=0, col=1)` isn't)

Parameters `row, col` (*int*, *int*) – row and col numbers localizing the entity among the result's table

Returns the partially initialized *Entity* instance

iter_rows_with_entities()

iterates over rows, and for each row eids are converted to plain entities

last(col=0)

Retrieve the last entity from the query.

If the result set is empty, raises `NoResultError`.

Parameters `col` (*int*) – The column localising the entity in the unique row

Returns the partially initialized *Entity* instance

limit(limit, offset=0, inplace=False)

limit the result set to the given number of rows optionally starting from an index different than 0

Parameters

- **limit** (*int*) – the maximum number of results
- **offset** (*int*) – the offset index
- **inplace** (*bool*) – if true, the result set is modified in place, else a new result set is returned and the original is left unmodified

Return type *ResultSet*

limited_rql()

returns a printable rql for the result set associated to the object, with limit/offset correctly set according to maximum page size and currently displayed page when necessary

one(col=0)

Retrieve exactly one entity from the query.

If the result set is empty, raises `NoResultError`. If the result set has more than one row, raises `MultipleResultsError`.

Parameters `col` (*int*) – The column localising the entity in the unique row

Returns the partially initialized *Entity* instance

possible_actions(kwargs)**

Return possible actions on this result set. Should always be called with the same arguments so it may be computed only once.

printable_rql()

return the result set's origin rql as a string, with arguments substituted

related_entity(kwargs)**

given an cell of the result set, try to return a (entity, relation name) tuple to which this cell is linked.

This is especially useful when the cell is an attribute of an entity, to get the entity to which this attribute belongs to.

searched_text(kwargs)**

returns the searched text in case of full-text search

Returns searched text or *None* if the query is not a full-text query

sorted_rset(keyfunc, reverse=False, col=0)

sorts the result set according to a given keyfunc

Parameters

- **keyfunc** (*callable(entity)*) – a callable which should take an entity as argument and return the value used to compare and sort
- **reverse** (*bool*) – if the result should be reversed
- **col** (*int*) – the column index. if col = -1, the whole row are used

Return type *ResultSet*

split_rset(keyfunc=None, col=0, return_dict=False)

splits the result set in multiple result sets according to a given key

Parameters

- **keyfunc** (*callable(entity or FinalType)*) – a callable which should take a value of the rset in argument and return the value used to group the value. If not define, raw value of the specified columns is used.
- **col** (*int*) – the column index. if col = -1, the whole row are used
- **return_dict** (*Boolean*) – If true, the function return a mapping (key -> rset) instead of a list of rset

Return type List of *ResultSet* or mapping of *ResultSet*

syntax_tree(kwargs)**

Return the **cached** syntax tree (`rql.stmts.Union`) for the originating query.

You can expect it to have solutions computed and it will be properly annotated. Since this is a cached shared object, **you must not modify it**.

transformed_rset(transformcb)

the result set according to a given column types

Parameters

- **transformcb** – a callable which should take a row and its type description as parameters, and return the transformed row and type description.
- **col** (*int*) – the column index

Return type *ResultSet*

11.10 cubicweb.web.views.urlpublishing

Associate url's path to view identifier / rql queries.

CubicWeb finds all registered `URLPathEvaluators`, orders them according to their `priority` attribute and calls their `evaluate_path()` method. The first that returns something and doesn't raise a `PathDontMatch` exception wins.

Here is the default evaluator chain:

1. `cubicweb.web.views.urlpublishing.RawPathEvaluator` handles unique url segments that match exactly one of the registered controller's `__regid__`. Urls such as `/view?`, `/edit?`, `/json?` fall in that category;
2. `cubicweb.web.views.urlpublishing.EidPathEvaluator` handles unique url segments that are eids (e.g. `/1234`);
3. `cubicweb.web.views.urlpublishing.URLRewriteEvaluator` selects all urlrewriter components, sorts them according to their priority, call their `rewrite()` method, the first one that doesn't raise a `KeyError` wins. This is where the `cubicweb.web.views.urlrewrite` and `cubicweb.web.views.urlrewrite.SimpleReqRewriter` comes into play;
4. `cubicweb.web.views.urlpublishing.RestPathEvaluator` handles urls based on entity types and attributes : `<type>((/<attribute name>)]?/<attribute value>)?` This is why `cwuser/carlos` works;
5. `cubicweb.web.views.urlpublishing.ActionPathEvaluator` handles any of the previous paths with an additional trailing `"/<action>"` segment, `<action>` being one of the registered actions' `__regid__`.

Note: Actionpath executes a query whose results is lost because of redirecting instead of direct traversal.

exception `cubicweb.web.views.urlpublishing.PathDontMatch`

exception used by url evaluators to notify they can't evaluate a path

class `cubicweb.web.views.urlpublishing.URLPublisherComponent` (*vreg*, *default_method*='view')

Bases: `cubicweb.view.Component`

Associate url path to view identifier / rql queries, by applying a chain of `urlpathevaluator` components.

An evaluator is a `URLPathEvaluator` subclass with an `.evaluate_path` method taking the request object and the path to publish as argument. It will either return a publishing method identifier and an rql query on success or raise a `PathDontMatch` exception on failure. URL evaluators are called according to their *priority* attribute, with 0 as the greatest priority and greater values as lower priority. The first evaluator returning a result or raising something else than `PathDontMatch` will stop the handlers chain.

process (*req*, *path*)

Given a URL (essentially characterized by a path on the server, but additional information may be found in the request object), return a publishing method identifier (e.g. controller) and an optional result set.

Parameters

- **req** (*cubicweb.web.request.CubicWebRequestBase*) – the request object
- **path** (*str*) – the path of the resource to publish. If empty, None or `"/"` `"view"` is used as the default path.

Return type `tuple(str, cubicweb.rset.ResultSet or None)`

Returns the publishing method identifier and an optional result set

Raises `NotFound` – if no handler is able to decode the given path

class cubicweb.web.views.urlpublishing.URLPathEvaluator(urlpublisher)

Bases: cubicweb.view.Component

class cubicweb.web.views.urlpublishing.RawPathEvaluator(urlpublisher)

Bases: [cubicweb.web.views.urlpublishing.URLPathEvaluator](#)

handle path of the form:

```
<publishing_method>?parameters...
```

class cubicweb.web.views.urlpublishing.EidPathEvaluator(urlpublisher)

Bases: [cubicweb.web.views.urlpublishing.URLPathEvaluator](#)

handle path with the form:

```
<eid>
```

class cubicweb.web.views.urlpublishing.RestPathEvaluator(urlpublisher)

Bases: [cubicweb.web.views.urlpublishing.URLPathEvaluator](#)

handle path with the form:

```
<etype>[[/<attribute name>]/<attribute value>]*
```

class cubicweb.web.views.urlpublishing.URLRewriteEvaluator(urlpublisher)

Bases: [cubicweb.web.views.urlpublishing.URLPathEvaluator](#)

tries to find a rewrite rule to apply

URL rewrite rule definitions are stored in URLRewriter objects

class cubicweb.web.views.urlpublishing.ActionPathEvaluator(urlpublisher)

Bases: [cubicweb.web.views.urlpublishing.URLPathEvaluator](#)

handle path with the form:

```
<any evaluator path>/<action>
```

11.11 cubicweb.web.views.urlrewrite

Rules based url rewriter component, to get configurable RESTful urls

class cubicweb.web.views.urlrewrite.URLRewriter(req, **extra)

Bases: [cubicweb.appobject.AppObject](#)

Base class for URL rewriters.

Url rewriters should have a *rules* dict that maps an input URI to something that should be used for rewriting.

The actual logic that defines how the rules dict is used is implemented in the *rewrite* method.

A *priority* attribute might be used to indicate which rewriter should be tried first. The higher the priority is, the earlier the rewriter will be tried.

class cubicweb.web.views.urlrewrite.**SimpleReqRewriter**(*req*, ***extra*)

Bases: [cubicweb.web.views.urlrewrite.URLRewriter](#)

The SimpleReqRewriters uses a *rules* dict that maps input URI (regexp or plain string) to a dictionary to update the request's form.

If the input uri is a regexp, group substitution is allowed.

rewrite(*req*, *uri*)

for each *input*, *output* in *rules*, if *uri* matches *input*, req's form is updated with *output*

class cubicweb.web.views.urlrewrite.**SchemaBasedRewriter**(*req*, ***extra*)

Bases: [cubicweb.web.views.urlrewrite.URLRewriter](#)

Here, the rules dict maps regexps or plain strings to callbacks that will be called with inputurl, uri, req, schema as parameters.

11.12 cubicweb.web

CubicWeb web client core. You'll need a apache-modpython or twisted publisher to get a full CubicWeb web application

11.12.1 Exceptions

exception cubicweb.web.**DirectResponse**(*response*)

Used to supply a twitted HTTP Response directly

exception cubicweb.web.**InvalidSession**

raised when a session id is found but associated session is not found or invalid

exception cubicweb.web.**PublishException**(**args*, ***kwargs*)

base class for publishing related exception

exception cubicweb.web.**Logout**(*url=None*)

raised to ask for deauthentication of a logged in user

exception cubicweb.web.**Redirect**(*location*, *status=HTTPStatus.SEE_OTHER*)

raised to redirect the http request

exception cubicweb.web.**StatusResponse**(*status*, *content=""*)

exception cubicweb.web.**RequestError**(**args*, ***kwargs*)

raised when a request can't be served because of a bad input

exception cubicweb.web.**NothingToEdit**(**args*, ***kwargs*)

raised when an edit request doesn't specify any eid to edit

exception cubicweb.web.**ProcessFormError**(**args*, ***kwargs*)

raised when posted data can't be processed by the corresponding field

exception cubicweb.web.**NotFound**(**args*, ***kwargs*)

raised when something was not found. In most case, a 404 error should be returned

exception cubicweb.web.**RemoteCallFailed**(*reason=""*, *status=HTTPStatus.INTERNAL_SERVER_ERROR*)

raised when a json remote call fails

***CUBICWEB* - THE SEMANTIC WEB IS A CONSTRUCTION GAME!**

CubicWeb is a semantic web application framework, licensed under the LGPL, empowering developers to efficiently build web applications by reusing components (called *cubes*) and following the well known object-oriented design principles.

12.1 Main Features

- an engine driven by the explicit *data model* of the application,
- a query language named *RQL* similar to W3C's SPARQL,
- a *selection+view* mechanism for semi-automatic XHTML/XML/JSON/text generation,
- a library of reusable *components* (data model and views) that fulfill common needs,
- the power and flexibility of the *Python* programming language,
- the reliability of SQL databases, LDAP directories and Mercurial for storage backends.

Created in early 2000s from an R&D effort and still maintained, supporting 100,000s of daily visits at some production sites, *CubicWeb* is a proven end to end solution for semantic web application development promoting quality, reusability and efficiency.

12.2 First steps

- **From scratch:**
 - *Install a CubicWeb environment*
 - *Configure a CubicWeb environment*
 - *Deploy a CubicWeb application*
- **Guides:**
 - *Introduction to CubicWeb*
 - Basics: *Building a simple blog with CubicWeb*
 - Advanced: *Building a photo gallery with CubicWeb*

12.3 Cubicweb core principle

- **Why cubicweb?**
 - *The Core Concepts of CubicWeb*
- **Cubes:**
 - *What is a Cube?*
 - *Creating a new cube from scratch*
 - *Standard structure for a cube*
- **Registries:**
 - *What are registries*
 - *How to use registries*
- **Data-centric framework:**
 - *Data schema with YAMS*
 - *Relation Query Language (RQL)*

12.4 Routing

CubicWeb offers two different ways of routing : one internal to CubicWeb and a one with the *pyramid framework*.

- **Principle:**
 - cubicweb and pyramid
 - the CW request object
 - *the pyramid request object*
 - encapsulation of the CW request in the pyramid request
 - `bw_compat` and the options to use, fallback when CW doesn't find anything
- **CubicWeb routing:**
 - *url publishers*
 - *url rewriters*
- **Pyramid routing:**
 - general principles
 - predicates
 - tweens
 - content negotiation

12.5 Front development

- **With Javascript / Typescript (using React):**
 - general principle
 - how to install and integrate js tooling into CW
 - cwelements
 - rql browser
- **With Pyramid:**
 - general integration with CubicWeb
 - [The renderers](#)
 - [Jinja2 templates](#)
 - example of usages with CW
- **With CubicWeb Views:**
 - [Introduction](#)
 - [Select a view with registers](#)
 - Facets
 - How to use javascript inside CW views
 - Customize CSS
- **RDF:**
 - the RDF adaptator
 - RDFLib integration into CW

12.6 Data model and management

- **Data in CubicWeb:**
 - [Data model](#)
 - [Data as objects](#)
- **Data Import:**
 - [Standard Import](#)
 - massive store

12.7 Security

- **Security:**
 - *The security model*
 - Permissions management with Pyramid
 - csrf_protection

12.8 Migrate your schema

Each time the schema is updated, two action are needed : update the underlying tables and update the corresponding data.

- **Migrations:**
 - *Execute and write migration script*
 - Debug script migration

12.9 Cubicweb configuration files

- **Base configuration:**
 - *The all-in-one.conf*
 - *The Pyramid configuration*
- **Advanced configuration:**
 - *The database connection pooler*

12.10 Common Web application tools

- **Test**
 - *CubicWeb*
 - Pyramid
- **Caching**
 - *HTTP cache management*
- **Internationalization**
 - *Localize your application*
- **Full text indexation**
 - *RQL search bar*

12.11 Development

- **Command line tool:**
 - *cubicweb-ctl tool*
- **Performances:**
 - *Profiling your application*
- **Debugging:**
 - Command line options for debugging
 - Debugging configuration directly in the code
 - Pyramid debug toolbar
 - Debug channels
- **Good practices:**
 - tox
 - check-manifest
 - mypy
 - flake8 et black
- **CI:**
 - Gitlab-ci integration

12.12 System administration

- **Deployment:**
 - *Raw python deployment*
 - *Working with Docker*
 - *Working with Kubernetes*
- **Administration:**
 - *Cubicweb-ctl tool*
 - Sources configuration
 - Backup

12.13 CubicWeb's ecosystem

CubicWeb is based on different libraries, in which you may be interested :

- [YAMS](#)
- [RQL](#)
- [logilab-common](#)
- [logilab-database](#)
- [logilab-constraints](#)
- [logilab-mtconverter](#)

12.14 How to contribute

- Chat on the [matrix room](#) `#cubicweb:matrix.logilab.org`
- Visio Weekly meeting every **Tuesday** afternoon (UTC+1). The link is shared in the [matrix room](#)
- Discuss on the [mailing-list](#)
- Discover on the [blog](#)
- Contribute on the [forge](#)
- Find published python modules on [pypi](#)
- Find published npm modules on [npm](#)
- *[Changelog](#)*

PYTHON MODULE INDEX

C

- cubicweb, 397
- cubicweb.appobject, 401
- cubicweb.cwconfig, 97
- cubicweb.cwvreg, 402
- cubicweb.dataimport, 410
- cubicweb.dataimport.importer, 191
- cubicweb.dataimport.massive_store, 194
- cubicweb.dataimport.stores, 192
- cubicweb.entities.adapters, 149
- cubicweb.predicates, 411
- cubicweb.pyramid, 417
- cubicweb.pyramid.auth, 418
- cubicweb.pyramid.bwcompat, 419
- cubicweb.pyramid.core, 419
- cubicweb.pyramid.defaults, 421
- cubicweb.pyramid.login, 422
- cubicweb.pyramid.session, 422
- cubicweb.pyramid.url_redirection, 424
- cubicweb.req, 424
- cubicweb.rset, 427
- cubicweb.rtags, 248
- cubicweb.server.hook, 163
- cubicweb.web, 432
- cubicweb.web.facet, 287
- cubicweb.web.formfields, 264
- cubicweb.web.formwidgets, 268
- cubicweb.web.httpcache, 296
- cubicweb.web.uihelper, 249
- cubicweb.web.views.ajaxcontroller, 250
- cubicweb.web.views.autoform, 258
- cubicweb.web.views.baseviews, 225
- cubicweb.web.views.formrenderers, 274
- cubicweb.web.views.forms, 272
- cubicweb.web.views.idownloadable, 247
- cubicweb.web.views.navigation, 227
- cubicweb.web.views.primary, 219
- cubicweb.web.views.startup, 229
- cubicweb.web.views.tableview, 230
- cubicweb.web.views.uicfg, 248
- cubicweb.web.views.urlpublishing, 242
- cubicweb.web.views.urlrewrite, 431

- cubicweb.web.views.wdoc, 247

- cubicweb.wfutils, 142

I

- logilab.common.registry, 405

Symbols

`__call__()` (*cubicweb.pyramid.bwcompat.CubicWebPyramidHandler* method), 419

`_cw_cnx()` (*in module cubicweb.pyramid.core*), 421

`_cw_request()` (*in module cubicweb.pyramid.core*), 421

`_cw_session()` (*in module cubicweb.pyramid.core*), 421

`_render()` (*cubicweb.web.formwidgets.FieldWidget* method), 268

-D
cubicweb-ctl-pyramid command line option, 300

--debug
cubicweb-ctl-pyramid command line option, 300

--debug-mode
cubicweb-ctl-pyramid command line option, 300

--loglevel
cubicweb-ctl-pyramid command line option, 301

--no-daemon
cubicweb-ctl-pyramid command line option, 300

--profile-dump-every
cubicweb-ctl-pyramid command line option, 301

--profile-output
cubicweb-ctl-pyramid command line option, 301

--reload
cubicweb-ctl-pyramid command line option, 300

--reload-interval
cubicweb-ctl-pyramid command line option, 301

-l
cubicweb-ctl-pyramid command line option, 301

A

`AboutAction` (*class in cubicweb.web.views.wdoc*), 248

`AbstractFacet` (*class in cubicweb.web.facet*), 291

`AbstractRangeRQLPathFacet` (*class in cubicweb.web.facet*), 290

`ActionPathEvaluator` (*class in cubicweb.web.views.urlpublishing*), 244, 431

`ActionsRegistry` (*class in cubicweb.cwvreg*), 405

`actual_fields()` (*cubicweb.web.formfields.Field* method), 265

`adaptable` (*class in cubicweb.predicates*), 411

`add_hidden()` (*cubicweb.web.form.FieldsForm.Form* method), 273

`add_operation()` (*cubicweb.repoapi.Connection* method), 207

`add_redirection_rule()` (*in module cubicweb.pyramid.url_redirection*), 424

`add_relation()` (*cubicweb.repoapi.Connection* method), 207

`add_relations()` (*cubicweb.repoapi.Connection* method), 207

`add_rql_restrictions()` (*cubicweb.web.facet.AbstractFacet* method), 291

`added_in_transaction()` (*cubicweb.repoapi.Connection* method), 207

`admin_request_from_url()` (*cubicweb.devtools.testlib.CubicWebTC* method), 179

`AjaxController` (*class in cubicweb.web.views.ajaxcontroller*), 250

`ajaxfunc()` (*in module cubicweb.web.views.ajaxcontroller*), 251

`AjaxFunction` (*class in cubicweb.web.views.ajaxcontroller*), 251

`AjaxWidget` (*class in cubicweb.web.formwidgets*), 271

`all()` (*cubicweb.rset.ResultSet* method), 427

`allow_all_hooks_but()` (*cubicweb.repoapi.Connection* method), 207

`allow_all_hooks_but()` (*cubicweb.server.session.Connection* method), 163, 207

`allowed_massmail_keys()` (*cubicweb.entities.adapters.IEmailableAdapter* method), 150

- AndPredicate (class in *logilab.common.registry*), 409
 anonymous_user (class in *cubicweb.predicates*), 415
 any_rset (class in *cubicweb.predicates*), 412
 AnyRsetView (class in *cubicweb.view*), 212
 app (*cubicweb.devtools.testlib.CubicWebTC* property), 180
 append_field() (*cubicweb.web.form.FieldsForm.Form* method), 273
 append_to_addmenu() (in module *cubicweb.web.uihelper*), 249
 AppObject (class in *cubicweb.appobject*), 401
 appobject_selectable (class in *cubicweb.predicates*), 411
 as_email_context() (*cubicweb.entities.adapters.IEmailableAdapter* method), 150
 assertSentEmail() (*cubicweb.devtools.testlib.CubicWebTC* method), 180
 attribute_edited (class in *cubicweb.predicates*), 416
 AttributeFacet (class in *cubicweb.web.facet*), 288
 attributes() (*cubicweb.web.formwidgets.FieldWidget* method), 269
 AttributeView (class in *cubicweb.web.views.primary*), 220
 authenticated_user (class in *cubicweb.predicates*), 415
 AuthenticationError, 398
 authors() (*cubicweb.entities.adapters.IDublinCoreAdapter* method), 149
 AutoCompletionWidget (class in *cubicweb.web.formwidgets*), 271
 AutomaticEntityForm (class in *cubicweb.web.views.autoform*), 258
- ## B
- BadCommandUsage, 399
 BadConnectionId, 398
 base_url() (*cubicweb.req.RequestSessionBase* method), 204, 424
 BigIntField (class in *cubicweb.web.formfields*), 266
 Binary (class in *cubicweb*), 400
 bind() (*cubicweb.web.views.tableview.EntityTableView.AbstractColumnRenderer* method), 233
 bind() (*cubicweb.web.views.tableview.PyValTableView.AbstractColumnRenderer* method), 234
 bind() (*cubicweb.web.views.tableview.RsetTableView.AbstractColumnRenderer* method), 231
 bind() (*cubicweb.web.views.tableview.TableMixIn.AbstractColumnRenderer* method), 236, 237
 BitFieldFacet (class in *cubicweb.web.facet*), 290
 BitSelect (class in *cubicweb.web.formwidgets*), 271
 bookmark_role() (in module *cubicweb.ext.rest*), 62
 BooleanField (class in *cubicweb.web.formfields*), 266
 breadcrumbs() (*cubicweb.web.views.ibreadcrumbs.IBreadCrumbsAdapter* method), 246
 build_column_renderers() (*cubicweb.web.views.pyviews.PyValTableView* method), 234
 build_column_renderers() (*cubicweb.web.views.tableview.EntityTableView* method), 233
 build_column_renderers() (*cubicweb.web.views.tableview.RsetTableView* method), 231
 build_column_renderers() (*cubicweb.web.views.tableview.TableMixIn* method), 236, 237
 build_suggestions() (*cubicweb.web.views.magicsearch.RQLSuggestionsBuilder* method), 210
 build_url() (*cubicweb.req.RequestSessionBase* method), 204, 424
 build_url_params() (*cubicweb.req.RequestSessionBase* method), 204, 425
 build_url_path() (*cubicweb.req.RequestSessionBase* method), 204, 425
 Button (class in *cubicweb.web.formwidgets*), 272
 ButtonInput (class in *cubicweb.web.formwidgets*), 269
- ## C
- cached_entities() (*cubicweb.repoapi.Connection* method), 207
 call() (*cubicweb.web.views.tableview.EntityTableView* method), 233
 call() (*cubicweb.web.views.tableview.RsetTableView* method), 231
 callfunc_every() (in module *cubicweb.dataimport*), 411
 CheckBox (class in *cubicweb.web.formwidgets*), 270
 children() (*cubicweb.entities.adapters.ITreeAdapter* method), 151
 children_rql() (*cubicweb.entities.adapters.ITreeAdapter* method), 151
 cleanupworkflow() (in module *cubicweb.wfutils*), 142
 clear() (*cubicweb.web.views.pyviews.PyValTableView* method), 234
 column_renderer() (*cubicweb.web.views.pyviews.PyValTableView* method), 238
 column_renderer() (*cubicweb.web.views.tableview.EntityTableView* method), 233
 column_renderer() (*cubicweb.web.views.tableview.RsetTableView* method), 231
 column_renderer() (*cubicweb.web.views.tableview.TableMixIn* method), 236, 237
 complete_entity() (*cubicweb.rset.ResultSet* method), 427
 CompositeForm (class in *cubicweb.web.views.forms*), 274

CompoundField (*class in cubicweb.web.formfields*), 267

compute_var_types() (*cubicweb.cwvreg.CWRegistryStore method*), 403

configcls (*cubicweb.devtools.testlib.CubicWebTC attribute*), 180

configuration value

- cubicweb.auth.authtgt (*bool*), 302
- cubicweb.auth.authtgt.persistent.cookie_name (*str*), 303
- cubicweb.auth.authtgt.persistent.max_age (*int*), 303
- cubicweb.auth.authtgt.persistent.reissue_time (*int*), 303
- cubicweb.auth.authtgt.persistent.samesite (*str*), 303
- cubicweb.auth.authtgt.session.cookie_name (*str*), 302
- cubicweb.auth.authtgt.session.reissue_time (*int*), 302
- cubicweb.auth.authtgt.session.samesite (*str*), 302
- cubicweb.auth.authtgt.session.timeout (*int*), 302
- cubicweb.auth.groups_principals (*bool*), 303
- cubicweb.auth.update_login_time (*bool*), 302
- cubicweb.bwcompat (*bool*), 302
- cubicweb.bwcompat.errorhandler (*bool*), 302
- cubicweb.debug (*bool*), 302
- cubicweb.defaults (*bool*), 302
- cubicweb.includes (*list*), 302
- cubicweb.instance (*string*), 302

configuration_values (*class in cubicweb.predicates*), 411

ConfigurationError, 399

Connection (*class in cubicweb.repoapi*), 206

Connection (*class in cubicweb.server.session*), 162

ConnectionCubicWebRequestBase (*class in cubicweb.web.request*), 210

ConnectionError, 398

content_navigation_components() (*cubicweb.web.views.primary.PrimaryView method*), 220

contextual (*class in cubicweb.predicates*), 416

core_handle() (*cubicweb.web.application.CubicWebPublisher method*), 200

count_lines() (*in module cubicweb.dataimport*), 411

create_entity() (*cubicweb.req.RequestSessionBase method*), 204, 425

create_user() (*cubicweb.devtools.testlib.CubicWebTC method*), 180

creator() (*cubicweb.entities.adapters.IDublinCoreAdapter method*), 149

critical() (*cubicweb.appobject.AppObject method*), 401

critical() (*cubicweb.repoapi.Connection method*), 208

critical() (*cubicweb.web.application.CubicWebPublisher method*), 200

CSVView (*class in cubicweb.web.views.baseviews*), 227

ctrl_publish() (*cubicweb.devtools.testlib.CubicWebTC method*), 180

ctxComponentsRegistry (*class in cubicweb.cwvreg*), 405

cubicweb

- module, 397
- cubicweb.appobject
- module, 401
- cubicweb.auth.authtgt (*bool*)
- configuration value, 302
- cubicweb.auth.authtgt.persistent.cookie_name (*str*)
- configuration value, 303
- cubicweb.auth.authtgt.persistent.max_age (*int*)
- configuration value, 303
- cubicweb.auth.authtgt.persistent.reissue_time (*int*)
- configuration value, 303
- cubicweb.auth.authtgt.persistent.samesite (*str*)
- configuration value, 303
- cubicweb.auth.authtgt.session.cookie_name (*str*)
- configuration value, 302
- cubicweb.auth.authtgt.session.reissue_time (*int*)
- configuration value, 302
- cubicweb.auth.authtgt.session.samesite (*str*)
- configuration value, 302
- cubicweb.auth.authtgt.session.timeout (*int*)
- configuration value, 302
- cubicweb.auth.groups_principals (*bool*)
- configuration value, 303
- cubicweb.auth.update_login_time (*bool*)
- configuration value, 302
- cubicweb.bwcompat (*bool*)
- configuration value, 302
- cubicweb.bwcompat.errorhandler (*bool*)
- configuration value, 302
- cubicweb.cwconfig
- module, 97
- cubicweb.cwvreg
- module, 402
- cubicweb.dataimport
- module, 410
- cubicweb.dataimport.importer
- module, 191
- cubicweb.dataimport.massive_store

- module, 194
- cubicweb.dataimport.stores
 - module, 192
- cubicweb.debug (*bool*)
 - configuration value, 302
- cubicweb.defaults (*bool*)
 - configuration value, 302
- cubicweb.entities.adapters
 - module, 149
- cubicweb.includes (*list*)
 - configuration value, 302
- cubicweb.instance (*string*)
 - configuration value, 302
- cubicweb.predicates
 - module, 411
- cubicweb.pyramid
 - module, 417
- cubicweb.pyramid.auth
 - module, 418
- cubicweb.pyramid.bwcompat
 - module, 419
- cubicweb.pyramid.core
 - module, 419
- cubicweb.pyramid.defaults
 - module, 421
- cubicweb.pyramid.login
 - module, 422
- cubicweb.pyramid.session
 - module, 422
- cubicweb.pyramid.url_redirection
 - module, 424
- cubicweb.req
 - module, 424
- cubicweb.rset
 - module, 427
- cubicweb.rtags
 - module, 248
- cubicweb.server.hook
 - module, 163
- cubicweb.web
 - module, 432
- cubicweb.web.facet
 - module, 287
- cubicweb.web.formfields
 - module, 264
- cubicweb.web.formwidgets
 - module, 268
- cubicweb.web.httppcache
 - module, 296
- cubicweb.web.uihelper
 - module, 249
- cubicweb.web.views.ajaxcontroller
 - module, 250
- cubicweb.web.views.autoform
 - module, 258
- cubicweb.web.views.baseviews
 - module, 225
- cubicweb.web.views.formrenderers
 - module, 274
- cubicweb.web.views.forms
 - module, 272
- cubicweb.web.views.idownloadable
 - module, 247
- cubicweb.web.views.navigation
 - module, 227
- cubicweb.web.views.primary
 - module, 219
- cubicweb.web.views.startup
 - module, 229
- cubicweb.web.views.tableview
 - module, 230
- cubicweb.web.views.uicfg
 - module, 248
- cubicweb.web.views.urlpublishing
 - module, 242, 430
- cubicweb.web.views.urlrewrite
 - module, 431
- cubicweb.web.views.wdoc
 - module, 247
- cubicweb.wfutils
 - module, 142
- cubicweb-ctl-pyramid command line option
 - D, 300
 - debug, 300
 - debug-mode, 300
 - loglevel, 301
 - no-daemon, 300
 - profile-dump-every, 301
 - profile-output, 301
 - reload, 300
 - reload-interval, 301
 - l, 301
- CubicWebEventManager (*class in cubicweb*), 400
- CubicWebException, 397
- CubicWebPublisher (*class in cubicweb.web.application*), 200
- CubicWebPyramidHandler (*class in cubicweb.pyramid.bwcompat*), 419
- CubicWebPyramidRequest (*class in cubicweb.pyramid.core*), 420
- CubicWebRuntimeError, 397
- CubicWebTC (*class in cubicweb.devtools.testlib*), 179
- CW_DEBUG, 417
- cw_fetch_order() (*cubicweb.entity.Entity class method*), 145
- cw_fetch_unrelated_order() (*cubicweb.entity.Entity class method*), 146

- `cw_fti_index_rql_limit()` (cubicweb.entities.AnyEntity class method), 189
- `CW_INSTANCE`, 417
- `CW_MODE`, 98
- `cw_propval()` (cubicweb.appobject.AppObject method), 401
- `cw_to_pyramid()` (in module cubicweb.pyramid.core), 419
- `CWRegistry` (class in cubicweb.cwvreg), 404
- `CWRegistryStore` (class in cubicweb.cwvreg), 402
- `CWSessionFactory()` (in module cubicweb.pyramid.session), 423
- `cwuri2eid()` (in module cubicweb.dataimport.importer), 192
- `CWUserRDFAdapter` (class in cubicweb.entities.adapters), 149
- ## D
- `DataOperationMixIn` (class in cubicweb.server.hook), 167
- `datapath()` (logilab.common.testlib.TestCase class method), 179
- `date()` (cubicweb.entities.adapters.IDublinCoreAdapter method), 149
- `DateField` (class in cubicweb.web.formfields), 266
- `DateRangeFacet` (class in cubicweb.web.facet), 290
- `DateRangeRQLPathFacet` (class in cubicweb.web.facet), 291
- `DateTimeField` (class in cubicweb.web.formfields), 267
- `DateTimePicker` (class in cubicweb.web.formwidgets), 270
- `DBG_ALL` (in module cubicweb.server), 337
- `DBG_HOOKS` (in module cubicweb.server), 337
- `DBG_MORE` (in module cubicweb.server), 337
- `DBG_NONE` (in module cubicweb.server), 337
- `DBG_OPS` (in module cubicweb.server), 337
- `DBG_REPO` (in module cubicweb.server), 337
- `DBG_RQL` (in module cubicweb.server), 337
- `DBG_SQL` (in module cubicweb.server), 337
- `debug()` (cubicweb.appobject.AppObject method), 401
- `debug()` (cubicweb.repoapi.Connection method), 208
- `debug()` (cubicweb.web.application.CubicWebPublisher method), 200
- `debug_mode` (class in cubicweb.predicates), 416
- `debugged` (class in cubicweb.server), 338
- `default_column_renderer_class` (cubicweb.web.views.pyviews.PyValTableView attribute), 235
- `default_column_renderer_class` (cubicweb.web.views.tableview.EntityTableView attribute), 233
- `default_column_renderer_class` (cubicweb.web.views.tableview.RsetTableView attribute), 231
- `delete_relation()` (cubicweb.repoapi.Connection method), 208
- `deleted_in_transaction()` (cubicweb.repoapi.Connection method), 208
- `deny_all_hooks_but()` (cubicweb.repoapi.Connection method), 208
- `deny_all_hooks_but()` (cubicweb.server.session.Connection method), 163, 207
- `description()` (cubicweb.entities.adapters.IDublinCoreAdapter method), 149
- `description_struct()` (cubicweb.rset.ResultSet method), 427
- `different_type_children()` (cubicweb.entities.adapters.ITreeAdapter method), 151
- `DirectResponse`, 432
- `display_func()` (cubicweb.web.views.navigation.SortedNavigation method), 228
- `dom_id()` (cubicweb.web.formfields.Field method), 265
- `domid` (cubicweb.web.views.pyviews.PyValTableView attribute), 235
- `domid` (cubicweb.web.views.tableview.TableMixIn attribute), 238
- `download_content_type()` (cubicweb.entities.adapters.IDownloadableAdapter method), 149
- `download_data()` (cubicweb.entities.adapters.IDownloadableAdapter method), 149
- `download_encoding()` (cubicweb.entities.adapters.IDownloadableAdapter method), 149
- `download_file_name()` (cubicweb.entities.adapters.IDownloadableAdapter method), 149
- `download_url()` (cubicweb.entities.adapters.IDownloadableAdapter method), 149
- `DownloadBox` (class in cubicweb.web.views.idownloadable), 247
- `DownloadLinkView` (class in cubicweb.web.views.idownloadable), 247
- `DownloadView` (class in cubicweb.web.views.idownloadable), 247
- `drop_entity_cache()` (cubicweb.repoapi.Connection method), 208
- ## E
- `edit_as_attr()` (in module cubicweb.web.uihelper), 249
- `edit_inline()` (in module cubicweb.web.uihelper), 249
- `EditableURLWidget` (class in cubicweb.web.formwidgets), 271

- EHTMLView (class in cubicweb.web.views.idownloadable), 247
- eid_reference_role() (in module cubicweb.ext.rest), 62
- eid_rset() (cubicweb.req.RequestSessionBase method), 204, 425
- EidNotInSource, 398
- EidPathEvaluator (class in cubicweb.web.views.urlpublishing), 243, 431
- EmailInput (class in cubicweb.web.formwidgets), 269
- empty_rset (class in cubicweb.predicates), 412
- empty_rset() (cubicweb.req.RequestSessionBase method), 204, 425
- ensure_ro_rql() (cubicweb.req.RequestSessionBase method), 204, 425
- entities() (cubicweb.rset.ResultSet method), 427
- entity() (cubicweb.web.views.tableview.EntityTableView method), 233
- entity() (cubicweb.web.views.tableview.EntityTableView.EntityTableColRender, 232, 425)
- entity() (cubicweb.web.views.tableview.EntityTableView.EntityTableColRender, 232, 425)
- entity_cache() (cubicweb.repoapi.Connection method), 208
- entity_from_eid() (cubicweb.req.RequestSessionBase method), 205, 425
- entity_sortvalue() (cubicweb.web.views.tableview.EntityTableView.EntityTableColRender, 232, 425)
- entity_type() (cubicweb.repoapi.Connection method), 208
- EntityCompositeFormRenderer (class in cubicweb.web.views.formrenderers), 274
- EntityFieldsForm (class in cubicweb.web.views.forms), 273
- EntityFormRenderer (class in cubicweb.web.views.formrenderers), 274
- EntityHTTPCacheManager (class in cubicweb.web.httpcache), 297
- EntityInlinedFormRenderer (class in cubicweb.web.views.formrenderers), 274
- EntityRDFAdapter (class in cubicweb.entities.adapters), 149
- EntityStartupView (class in cubicweb.view), 212
- EntityTableView (class in cubicweb.web.views.tableview), 232
- EntityTableView.AbstractColumnRenderer (class in cubicweb.web.views.tableview), 233
- EntityTableView.EntityTableColRenderer (class in cubicweb.web.views.tableview), 232
- EntityTableView.MainEntityColRenderer (class in cubicweb.web.views.tableview), 232
- EntityTableView.RelatedEntityColRenderer (class in cubicweb.web.views.tableview), 232
- EntityTableView.RelationColRenderer (class in cubicweb.web.views.tableview), 232
- EntityView (class in cubicweb.view), 212
- environment variable
- CW_DEBUG, 417
 - CW_INSTANCE, 417
 - CW_INSTANCES_DATA_DIR, 99
 - CW_INSTANCES_DIR, 99
 - CW_MODE, 98, 99
 - CW_RUNTIME_DIR, 99
- error() (cubicweb.appobject.AppObject method), 402
- error() (cubicweb.repoapi.Connection method), 208
- error() (cubicweb.web.application.CubicWebPublisher method), 200
- EtagHTTPCacheManager (class in cubicweb.web.httpcache), 297
- etype_class() (cubicweb.cwvreg.ETypeRegistry method), 404
- etype_rset() (cubicweb.req.RequestSessionBase method), 204, 425
- ETypeRegistry (class in cubicweb.cwvreg), 404
- exception() (cubicweb.appobject.AppObject method), 402
- exception() (cubicweb.repoapi.Connection method), 208
- exception() (cubicweb.web.application.CubicWebPublisher method), 200
- execute() (cubicweb.web.views.tableview.EntityTableView.EntityTableColRender, 232, 425)
- ExecutionError, 399
- exists() (cubicweb.req.RequestSessionBase method), 205, 425
- expect_redirect() (cubicweb.devtools.testlib.CubicWebTC method), 180
- expect_redirect_handle_request() (cubicweb.devtools.testlib.CubicWebTC method), 180
- ExtEntitiesImporter (class in cubicweb.dataimport.importer), 191
- ExtEntity (class in cubicweb.dataimport.importer), 192
- ## F
- fake_form() (cubicweb.devtools.testlib.CubicWebTC static method), 180
- FCKEditor (class in cubicweb.web.formwidgets), 271
- fetch_attrs() (cubicweb.cwvreg.ETypeRegistry method), 405
- fetch_config() (in module cubicweb.entities), 145
- Field (class in cubicweb.web.formfields), 264
- field_by_name() (cubicweb.web.form.FieldsForm.Form method), 273
- fields_by_name() (cubicweb.web.form.FieldsForm.Form method), 273

- FieldsForm (class in *cubicweb.web.views.forms*), 272
 FieldWidget (class in *cubicweb.web.formwidgets*), 268
 FileField (class in *cubicweb.web.formfields*), 267
 FileInput (class in *cubicweb.web.formwidgets*), 269
 filtered_rset() (*cubicweb.rset.ResultSet* method), 427
 FinalView (class in *cubicweb.web.views.baseviews*), 225
 find() (*cubicweb.req.RequestSessionBase* method), 205, 425
 first() (*cubicweb.rset.ResultSet* method), 427
 FloatField (class in *cubicweb.web.formfields*), 266
 Forbidden, 398
 form_init() (*cubicweb.web.formfields.Field* method), 265
 format_date() (*cubicweb.req.RequestSessionBase* method), 205, 425
 format_float() (*cubicweb.req.RequestSessionBase* method), 205, 425
 format_link_content() (*cubicweb.web.views.navigation.SortedNavigation* method), 228
 format_time() (*cubicweb.req.RequestSessionBase* method), 205, 425
 FormConfig (class in *cubicweb.web.uihelper*), 249
 FormRenderer (class in *cubicweb.web.views.formrenderers*), 274
 fti_containers() (*cubicweb.entities.adapters.IFTIndexableAdapter* method), 150
- ## G
- get_entity() (*cubicweb.rset.ResultSet* method), 427
 get_option_value() (*cubicweb.repoapi.Connection* method), 208
 get_principals() (in module *cubicweb.pyramid.core*), 420
 get_schema() (*cubicweb.repoapi.Connection* method), 208
 get_session() (*cubicweb.web.application.CubicWebPublisher* method), 200
 get_widget() (*cubicweb.web.facet.AbstractFacet* method), 291
 get_words() (*cubicweb.entities.adapters.IFTIndexableAdapter* method), 150
 guess_field() (in module *cubicweb.web.formfields*), 268
- ## H
- has_add_permission (class in *cubicweb.predicates*), 414
 has_headers (*cubicweb.web.views.pyviews.PyValTableView* property), 235
 has_mimetype (class in *cubicweb.predicates*), 414
 has_permission (class in *cubicweb.predicates*), 414
 has_related_entities (class in *cubicweb.predicates*), 414
 HasRelationFacet (class in *cubicweb.web.facet*), 288
 HelpAction (class in *cubicweb.web.views.wdoc*), 248
 HiddenInput (class in *cubicweb.web.formwidgets*), 269
 hide_field() (in module *cubicweb.web.uihelper*), 249
 hide_fields() (in module *cubicweb.web.uihelper*), 249
 Hook (class in *cubicweb.server.hook*), 166
 HorizontalLayoutWidget (class in *cubicweb.web.formwidgets*), 271
 HTableFormRenderer (class in *cubicweb.web.views.formrenderers*), 274
 http_method() (*cubicweb.pyramid.core.CubicWebPyramidRequest* method), 420
 http_publish() (*cubicweb.devtools.testlib.CubicWebTC* method), 180
- ## I
- IDownloadableAdapter (class in *cubicweb.entities.adapters*), 149
 IDownloadableOneLineView (class in *cubicweb.web.views.idownloadable*), 247
 IDownloadablePrimaryView (class in *cubicweb.web.views.idownloadable*), 247
 IDublinCoreAdapter (class in *cubicweb.entities.adapters*), 149
 IEmailableAdapter (class in *cubicweb.entities.adapters*), 150
 IFTIndexableAdapter (class in *cubicweb.entities.adapters*), 150
 ImageView (class in *cubicweb.web.views.idownloadable*), 247
 ImgButton (class in *cubicweb.web.formwidgets*), 272
 import_entities() (*cubicweb.dataimport.importer.ExtEntitiesImporter* method), 191
 includeme() (in module *cubicweb.pyramid.auth*), 418
 includeme() (in module *cubicweb.pyramid.bwcompat*), 419
 includeme() (in module *cubicweb.pyramid.core*), 419
 includeme() (in module *cubicweb.pyramid.defaults*), 421
 includeme() (in module *cubicweb.pyramid.login*), 422
 includeme() (in module *cubicweb.pyramid.session*), 423
 includeme() (in module *cubicweb.pyramid.url_redirection*), 424
 InContextTextView (class in *cubicweb.web.views.baseviews*), 227
 InContextView (class in *cubicweb.web.views.baseviews*), 226
 IndexView (class in *cubicweb.web.views.startup*), 229
 info() (*cubicweb.appobject.AppObject* method), 402
 info() (*cubicweb.repoapi.Connection* method), 208

`info()` (*cubicweb.web.application.CubicWebPublisher* method), 200

`init_config()` (*cubicweb.devtools.testlib.CubicWebTC* class method), 180

`init_properties()` (*cubicweb.cwvreg.CWRegistryStore* method), 403

`initial_load` (*cubicweb.web.views.tableview.TableLayout* attribute), 236

`initialization_completed()` (*cubicweb.cwvreg.CWRegistryStore* method), 403

`initialization_completed()` (*cubicweb.cwvreg.ETypeRegistry* method), 405

`InlineHelpView` (class in *cubicweb.web.views.wdoc*), 247

`innerSkip()` (*logilab.common.testlib.TestCase* method), 179

`INotifiableAdapter` (class in *cubicweb.entities.adapters*), 150

`InOutWidget` (class in *cubicweb.web.formwidgets*), 271

`input_name()` (*cubicweb.web.formfields.Field* method), 265

`insert_field_after()` (*cubicweb.web.form.FieldsForm.Form* method), 273

`insert_field_before()` (*cubicweb.web.form.FieldsForm.Form* method), 273

`InstancesRegistry` (class in *cubicweb.cwvreg*), 404

`InternalError`, 397

`IntervalWidget` (class in *cubicweb.web.formwidgets*), 271

`IntField` (class in *cubicweb.web.formfields*), 266

`InvalidSession`, 432

`IPrevNextAdapter` (class in *cubicweb.web.views.navigation*), 229

`is_hook_activated()` (*cubicweb.repoapi.Connection* method), 209

`is_hook_category_activated()` (*cubicweb.repoapi.Connection* method), 209

`is_in_state` (class in *cubicweb.predicates*), 414

`is_instance` (class in *cubicweb.predicates*), 412

`is_leaf()` (*cubicweb.entities.adapters.ITreeAdapter* method), 151

`is_reload_needed()` (*cubicweb.cwvreg.CWRegistryStore* method), 403

`is_root()` (*cubicweb.entities.adapters.ITreeAdapter* method), 151

`ISerializableAdapter` (class in *cubicweb.entities.adapters*), 150

`items()` (*cubicweb.cwvreg.CWRegistryStore* method), 403

`iter_modified_fields()` (*cubicweb.web.views.forms.FieldsForm* method), 273

`iter_rows_with_entities()` (*cubicweb.rset.ResultSet* method), 428

`iterchildren()` (*cubicweb.entities.adapters.ITreeAdapter* method), 151

`iteritems()` (*cubicweb.cwvreg.CWRegistryStore* method), 403

`iterparents()` (*cubicweb.entities.adapters.ITreeAdapter* method), 151

`itervalues()` (*cubicweb.cwvreg.CWRegistryStore* method), 403

`ITreeAdapter` (class in *cubicweb.entities.adapters*), 150

`IUserFriendlyCheckConstraint` (class in *cubicweb.entities.adapters*), 151

`IUserFriendlyError` (class in *cubicweb.entities.adapters*), 151

`IUserFriendlyUniqueTogether` (class in *cubicweb.entities.adapters*), 151

J

`jQueryDatePicker` (class in *cubicweb.web.formwidgets*), 270

`jQueryDateTimePicker` (class in *cubicweb.web.formwidgets*), 270

`jQueryTimePicker` (class in *cubicweb.web.formwidgets*), 270

L

`language()` (*cubicweb.entities.adapters.IDublinCoreAdapter* method), 149

`last()` (*cubicweb.rset.ResultSet* method), 428

`LateOperation` (class in *cubicweb.server.hook*), 167

`layout_args` (*cubicweb.web.views.pyviews.PyValTableView* property), 235

`limit()` (*cubicweb.rset.ResultSet* method), 428

`limited_rql()` (*cubicweb.rset.ResultSet* method), 428

`linkable()` (*cubicweb.web.views.tableview.RsetTableView* method), 231

`list_actions_for()` (*cubicweb.devtools.testlib.CubicWebTC* method), 180

`list_boxes_for()` (*cubicweb.devtools.testlib.CubicWebTC* method), 180

`list_startup_views()` (*cubicweb.devtools.testlib.CubicWebTC* method), 181

`list_views_for()` (*cubicweb.devtools.testlib.CubicWebTC* method), 181

`ListView` (class in *cubicweb.web.views.baseviews*), 226

[load_file\(\)](#) (*cubicweb.cwvreg.CWRegistryStore* method), 403
[locate_all_files\(\)](#) (*cubicweb.web.webconfig.WebConfiguration* method), 297
[locate_doc_file\(\)](#) (*cubicweb.web.webconfig.WebConfiguration* method), 297
[locate_resource\(\)](#) (*cubicweb.web.webconfig.WebConfiguration* method), 297
[log_handle_request\(\)](#) (*cubicweb.web.application.CubicWebPublisher* method), 200
[logilab.common.registry](#) module, 405
[login_already_loggedin\(\)](#) (in module *cubicweb.pyramid.login*), 422
[login_form\(\)](#) (in module *cubicweb.pyramid.login*), 422
[login_password_login\(\)](#) (in module *cubicweb.pyramid.login*), 422
[Logout](#), 432
[long_title\(\)](#) (*cubicweb.entities.adapters.IDublinCoreAdapter* method), 149

M

[main_handle_request\(\)](#) (*cubicweb.web.application.CubicWebPublisher* method), 200, 201
[main_template\(\)](#) (*cubicweb.cwvreg.ViewsRegistry* method), 405
[main_var_index\(\)](#) (*cubicweb.web.views.tableview.RsetTableView* method), 232
[ManageView](#) (class in *cubicweb.web.views.startup*), 229
[MassiveObjectStore](#) (class in *cubicweb.dataimport.massive_store*), 194
[match_context](#) (class in *cubicweb.predicates*), 416
[match_context_prop](#) (class in *cubicweb.predicates*), 415
[match_exception](#) (class in *cubicweb.predicates*), 416
[match_form_params](#) (class in *cubicweb.predicates*), 415
[match_kwargs](#) (class in *cubicweb.predicates*), 411
[match_rtype](#) (class in *cubicweb.server.hook*), 166
[match_rtype_sets](#) (class in *cubicweb.server.hook*), 166
[match_search_state](#) (class in *cubicweb.predicates*), 415
[match_transition](#) (class in *cubicweb.predicates*), 416
[match_user_groups](#) (class in *cubicweb.predicates*), 415
[match_view](#) (class in *cubicweb.predicates*), 416
[MaxAgeHTTPCacheManager](#) (class in *cubicweb.web.httppcache*), 297
[maxDiff](#) (*logilab.common.testlib.TestCase* attribute), 179
[message](#) (*cubicweb.pyramid.core.CubicWebPyramidRequest* property), 420
[MetadataGenerator](#) (class in *cubicweb.dataimport.stores*), 193
[module](#)
 [cubicweb](#), 397
 [cubicweb.appobject](#), 401
 [cubicweb.cwconfig](#), 97
 [cubicweb.cwvreg](#), 402
 [cubicweb.dataimport](#), 410
 [cubicweb.dataimport.importer](#), 191
 [cubicweb.dataimport.massive_store](#), 194
 [cubicweb.dataimport.stores](#), 192
 [cubicweb.entities.adapters](#), 149
 [cubicweb.predicates](#), 411
 [cubicweb.pyramid](#), 417
 [cubicweb.pyramid.auth](#), 418
 [cubicweb.pyramid.bwcompat](#), 419
 [cubicweb.pyramid.core](#), 419
 [cubicweb.pyramid.defaults](#), 421
 [cubicweb.pyramid.login](#), 422
 [cubicweb.pyramid.session](#), 422
 [cubicweb.pyramid.url_redirection](#), 424
 [cubicweb.req](#), 424
 [cubicweb.rset](#), 427
 [cubicweb.rtags](#), 248
 [cubicweb.server.hook](#), 163
 [cubicweb.web](#), 432
 [cubicweb.web.facet](#), 287
 [cubicweb.web.formfields](#), 264
 [cubicweb.web.formwidgets](#), 268
 [cubicweb.web.httppcache](#), 296
 [cubicweb.web.uihelper](#), 249
 [cubicweb.web.views.ajaxcontroller](#), 250
 [cubicweb.web.views.autoform](#), 258
 [cubicweb.web.views.baseviews](#), 225
 [cubicweb.web.views.formrenderers](#), 274
 [cubicweb.web.views.forms](#), 272
 [cubicweb.web.views.idownloadable](#), 247
 [cubicweb.web.views.navigation](#), 227
 [cubicweb.web.views.primary](#), 219
 [cubicweb.web.views.startup](#), 229
 [cubicweb.web.views.tableview](#), 230
 [cubicweb.web.views.uicfg](#), 248
 [cubicweb.web.views.urlpublishing](#), 242, 430
 [cubicweb.web.views.urlrewrite](#), 431
 [cubicweb.web.views.wdoc](#), 247
 [cubicweb.wfutils](#), 142
 [logilab.common.registry](#), 405
[multi_columns_rset](#) (class in *cubicweb.predicates*), 412
[multi_etypes_rset](#) (class in *cubicweb.predicates*), 412
[multi_lines_rset](#) (class in *cubicweb.predicates*), 412
[MultipleResultsError](#), 399

N

`new_access()` (*cubicweb.devtools.testlib.CubicWebTC* method), 181

`next_entity()` (*cubicweb.web.views.navigation.IPrevNextAdapter* method), 229

`NextPrevNavigationComponent` (class in *cubicweb.web.views.navigation*), 229

`no_cnx` (class in *cubicweb.predicates*), 415

`NoEtag`, 297

`NoHookRQLObjStore` (class in *cubicweb.dataimport.stores*), 193

`NoHookRQLObjStore` (in module *cubicweb.dataimport*), 411

`NoHTTPCacheManager` (class in *cubicweb.web.httpcache*), 297

`non_final_entity` (class in *cubicweb.predicates*), 412

`none_rset` (class in *cubicweb.predicates*), 412

`nonempty_rset` (class in *cubicweb.predicates*), 412

`NoResultError`, 399

`NoResultView` (class in *cubicweb.web.views.baseviews*), 225

`NoSelectableObject` (class in *logilab.common.registry*), 410

`NotAnEntity`, 399

`NotFound`, 432

`NothingToEdit`, 432

`notification_references()` (*cubicweb.entities.adapters.INotifiableAdapter* method), 150

`NotificationView` (class in *cubicweb.subjects.notification*), 172

`notify_on_commit()` (in module *cubicweb.hooks.notification*), 172

`NotPredicate` (class in *logilab.common.registry*), 409

`NullStore` (class in *cubicweb.dataimport.stores*), 193

`NullView` (class in *cubicweb.web.views.baseviews*), 225

O

`object_by_id()` (*logilab.common.registry.Registry* method), 408

`objectify_predicate()` (in module *logilab.common.registry*), 409

`ObjectNotFound` (class in *logilab.common.registry*), 410

`on_fire_transition()` (in module *cubicweb.predicates*), 415

`one()` (*cubicweb.rset.ResultSet* method), 428

`one_etype_rset` (class in *cubicweb.predicates*), 412

`one_line_rset` (class in *cubicweb.predicates*), 412

`OneLineView` (class in *cubicweb.web.views.baseviews*), 226

`onevent()` (in module *cubicweb*), 400

`Operation` (class in *cubicweb.server.hook*), 166

`operator` (*cubicweb.web.facet.AbstractFacet* attribute), 291

`optval()` (*logilab.common.testlib.TestCase* method), 179

`OrPredicate` (class in *logilab.common.registry*), 409

`OutOfContextView` (class in *cubicweb.web.views.baseviews*), 226, 227

P

`PageNavigation` (class in *cubicweb.web.views.navigation*), 228

`PageNavigationSelect` (class in *cubicweb.web.views.navigation*), 228

`paginate()` (in module *cubicweb.web.views.navigation*), 229

`paginated_rset` (class in *cubicweb.predicates*), 412

`parent()` (*cubicweb.entities.adapters.ITreeAdapter* method), 151

`parent_entity()` (*cubicweb.web.views.ibreadcrumbs.IBreadCrumbsAdapter* method), 246

`parse()` (*cubicweb.cwvreg.CWRegistryStore* method), 403

`parse_datetime()` (*cubicweb.req.RequestSessionBase* method), 205, 426

`partial_has_related_entities` (class in *cubicweb.predicates*), 414

`partial_relation_possible` (class in *cubicweb.predicates*), 414

`PasswordField` (class in *cubicweb.web.formfields*), 266

`PasswordInput` (class in *cubicweb.web.formwidgets*), 271

`PasswordSingleInput` (class in *cubicweb.web.formwidgets*), 269

`path()` (*cubicweb.entities.adapters.ITreeAdapter* method), 151

`PathDontMatch`, 430

`pending_operations` (*cubicweb.repoapi.Connection* attribute), 209

`poss_visible_objects()` (*cubicweb.cwvreg.ActionsRegistry* method), 405

`poss_visible_objects()` (*cubicweb.cwvreg.CtxComponentsRegistry* method), 405

`poss_visible_objects()` (*cubicweb.cwvreg.CWRegistry* method), 404

`possible_actions()` (*cubicweb.rset.ResultSet* method), 428

`possible_objects()` (*logilab.common.registry.Registry* method), 408

`possible_values()` (*cubicweb.web.facet.VocabularyFacet* method), 291

possible_views() (*cubicweb.cwvreg.ViewsRegistry* method), 405
 pre_setup_database() (*cubicweb.devtools.testlib.CubicWebTC* class method), 181
 Predicate (class in *logilab.common.registry*), 409
 prefixiter() (*cubicweb.entities.adapters.ITreeAdapter* method), 151
 previous_entity() (*cubicweb.web.views.navigation.IPrevNextAdapter* method), 229
 primary_view (class in *cubicweb.predicates*), 416
 PrimaryView (class in *cubicweb.web.views.primary*), 219
 printable_rql() (*cubicweb.rset.ResultSet* method), 428
 printable_value() (*cubicweb.req.RequestSessionBase* method), 205, 426
 process() (*cubicweb.web.views.urlpublishing.URLPublisherComponent* method), 242, 430
 process_field_data() (*cubicweb.web.formwidgets.FieldWidget* method), 269
 process_form_value() (*cubicweb.web.formfields.Field* method), 266
 process_posted() (*cubicweb.web.formfields.Field* method), 266
 process_posted() (*cubicweb.web.views.forms.FieldsForm* method), 273
 ProcessFormError, 432
 ProgrammingError, 397
 property_info() (*cubicweb.cwvreg.CWRegistryStore* method), 403
 property_value() (*cubicweb.cwvreg.CWRegistryStore* method), 403
 property_value() (*cubicweb.req.RequestSessionBase* method), 205, 426
 publish() (*cubicweb.web.views.ajaxcontroller.AjaxController* method), 251
 PublishException, 432
 PyramidSessionHandler (class in *cubicweb.pyramid.bwcompat*), 419
 PyValTableView (class in *cubicweb.web.views.pyviews*), 234
 PyValTableView.AbstractColumnRenderer (class in *cubicweb.web.views.tableview*), 234
 RangeFacet (class in *cubicweb.web.facet*), 290
 RangeRQLPathFacet (class in *cubicweb.web.facet*), 290
 RawPathEvaluator (class in *cubicweb.web.views.urlpublishing*), 243, 431
 rebuild_url() (*cubicweb.req.RequestSessionBase* method), 205, 426
 Redirect, 432
 redirect_handler() (*cubicweb.web.application.CubicWebPublisher* method), 201
 register() (*cubicweb.cwvreg.CWRegistryStore* method), 403
 register() (*cubicweb.cwvreg.ETypeRegistry* method), 405
 register() (*logilab.common.registry.Registry* method), 408
 register() (*logilab.common.registry.RegistryStore* method), 407
 register_all() (*cubicweb.cwvreg.CWRegistryStore* method), 403
 register_all() (*logilab.common.registry.RegistryStore* method), 407
 register_and_replace() (*cubicweb.cwvreg.CWRegistryStore* method), 403
 register_and_replace() (*logilab.common.registry.RegistryStore* method), 407
 register_modnames() (*logilab.common.registry.RegistryStore* method), 407
 register_property() (*cubicweb.cwvreg.CWRegistryStore* method), 403
 Registry (class in *logilab.common.registry*), 408
 REGISTRY_FACTORY (*cubicweb.cwvreg.CWRegistryStore* attribute), 402
 RegistryException (class in *logilab.common.registry*), 410
 RegistryNotFound (class in *logilab.common.registry*), 410
 RegistryStore (class in *logilab.common.registry*), 406
 related_entity() (*cubicweb.rset.ResultSet* method), 428
 relation_possible (class in *cubicweb.predicates*), 413
 RelationAttributeFacet (class in *cubicweb.web.facet*), 288
 RelationFacet (class in *cubicweb.web.facet*), 287
 RelationField (class in *cubicweb.web.formfields*), 268
 RelationMapping (class in *cubicweb.dataimport.importer*), 192
 relative_path() (*cubicweb.pyramid.core.CubicWebPyramidRequest*

Q

QueryError, 399

R

Radio (class in *cubicweb.web.formwidgets*), 270

`method`), 420
`reload()` (*cubicweb.cwvreg.CWRegistryStore method*), 403
`reload_if_needed()` (*cubicweb.cwvreg.CWRegistryStore method*), 404
`remote_calling()` (*cubicweb.devtools.testlib.CubicWebTC method*), 181
`RemoteCallFailed`, 432
`remove_field()` (*cubicweb.web.form.FieldsForm.Form method*), 273
`remove_from_addmenu()` (*in module cubicweb.web.uihelper*), 249
`render()` (*cubicweb.web.formwidgets.FieldWidget method*), 268
`render()` (*cubicweb.web.views.forms.FieldsForm method*), 273
`render()` (*cubicweb.web.views.tableview.TableLayout method*), 236
`render_cell()` (*cubicweb.web.views.tableview.EntityTableView method*), 233
`render_cell()` (*cubicweb.web.views.tableview.PyValTableView method*), 234
`render_cell()` (*cubicweb.web.views.tableview.RsetTableView method*), 231
`render_cell()` (*cubicweb.web.views.tableview.TableMixIn method*), 236, 237
`render_entity()` (*cubicweb.web.views.tableview.EntityTableView.EntityTableColumnRenderer method*), 232
`render_entity_attributes()` (*cubicweb.web.views.primary.PrimaryView method*), 219
`render_entity_relations()` (*cubicweb.web.views.primary.PrimaryView method*), 219
`render_entity_title()` (*cubicweb.web.views.primary.PrimaryView method*), 219
`render_header()` (*cubicweb.web.views.tableview.EntityTableView.AbstractColumnRenderer method*), 233
`render_header()` (*cubicweb.web.views.tableview.PyValTableView.AbstractColumnRenderer method*), 234
`render_header()` (*cubicweb.web.views.tableview.RsetTableView.AbstractColumnRenderer method*), 231
`render_header()` (*cubicweb.web.views.tableview.TableMixIn.AbstractColumnRenderer method*), 236, 237
`render_side_boxes()` (*cubicweb.web.views.primary.PrimaryView method*), 219
`render_view()` (*in module cubicweb.pyramid.core*), 420
`repo` (*cubicweb.repoapi.Connection attribute*), 209
`repo_connect()` (*in module cubicweb.pyramid.core*), 420
`RepositoryError`, 397
`requestcls` (*cubicweb.devtools.testlib.CubicWebTC attribute*), 181
`RequestError`, 432
`RequestSessionBase` (*class in cubicweb.req*), 204, 424
`reset()` (*cubicweb.cwvreg.CWRegistryStore method*), 404
`ResetButton` (*class in cubicweb.web.formwidgets*), 272
`RestPathEvaluator` (*class in cubicweb.web.views.urlpublishing*), 244, 431
`ResultSet` (*class in cubicweb.rset*), 427
`rewrite()` (*cubicweb.web.views.urlrewrite.SimpleReqRewriter method*), 244, 432
`RichString()` (*in module yams.buildobjs*), 126
`RichTextField` (*class in cubicweb.web.formfields*), 267
`RQLAdapter` (*class in cubicweb.rest.abstradapter*), 151
`rql_adapter()` (*in module cubicweb.ext.rest*), 62
`rql_role()` (*in module cubicweb.ext.rest*), 62
`RQObjectStore` (*class in cubicweb.dataimport.stores*), 193
`RQObjectStore` (*in module cubicweb.dataimport*), 411
`RQLPathFacet` (*class in cubicweb.web.facet*), 289
`RQLSuggestionsBuilder` (*class in cubicweb.web.views.magicsearch*), 210
`RsetTableView` (*class in cubicweb.web.views.tableview*), 230
`RsetTableView.AbstractColumnRenderer` (*class in cubicweb.web.views.tableview*), 231
`RsetTableView.RsetTableColRenderer` (*class in cubicweb.web.views.tableview*), 231
`running_hooks_ops()` (*cubicweb.repoapi.Connection method*), 209

S

`same_type_children()` (*cubicweb.entities.adapters.FreeAdapter method*), 151
`SameTypeListView` (*class in cubicweb.web.views.base_views*), 227
`schema` (*cubicweb.cwvreg.CWRegistry property*), 404
`schema` (*cubicweb.devtools.testlib.CubicWebTC property*), 181

S

452

schema: created_by owned_by; is; is_instance; 138
 schema: eid creation_date; modification_data; cwuri, 138
 schema: meta-data; 138
 SchemaBasedRewriter (class in cubicweb.web.views.urlrewrite), 244, 432
 score_entity (class in cubicweb.predicates), 413
 searched_text() (cubicweb.rset.ResultSet method), 429
 SecurityError, 397
 Select (class in cubicweb.web.formwidgets), 270
 select() (cubicweb.cwvreg.CWRegistry method), 404
 select() (logilab.common.registry.Registry method), 408
 select_or_none() (logilab.common.registry.Registry method), 408
 selected() (cubicweb.cwvreg.InstancesRegistry method), 404
 set_debug() (in module cubicweb.server), 337
 set_description() (logilab.common.testlib.TestCase method), 179
 set_entity_cache() (cubicweb.repoapi.Connection method), 209
 set_field() (in module cubicweb.web.uihelper), 249
 set_field_kwargs() (in module cubicweb.web.uihelper), 249
 set_fields_order() (in module cubicweb.web.uihelper), 249
 set_http_cache_headers() (in module cubicweb.web.httpcache), 297
 set_language() (cubicweb.req.RequestSessionBase method), 206, 426
 set_muledit_editable() (in module cubicweb.web.uihelper), 249
 set_schema() (cubicweb.cwvreg.CWRegistryStore method), 404
 setdefault() (cubicweb.cwvreg.CWRegistryStore method), 404
 setUp() (cubicweb.devtools.testlib.CubicWebTC method), 181
 setup_database() (cubicweb.devtools.testlib.CubicWebTC method), 181
 setup_params() (cubicweb.pyramid.core.CubicWebPyramid method), 421
 setup_workflow() (in module cubicweb.wfutils), 142
 setUpClass() (cubicweb.devtools.testlib.CubicWebTC class method), 181
 shortDescription() (logilab.common.testlib.TestCase method), 179
 SimpleListView (class in cubicweb.web.views.baseviews), 226
 SimpleReqRewriter (class in cubicweb.web.views.urlrewrite), 244, 431
 sort_on() (cubicweb.web.views.navigation.SortedNavigation method), 228
 sorted_rset (class in cubicweb.predicates), 412
 sorted_rset() (cubicweb.rset.ResultSet method), 429
 SortedNavigation (class in cubicweb.web.views.navigation), 228
 sortvalue() (cubicweb.web.views.tableview.EntityTableView.AbstractColumn method), 233
 sortvalue() (cubicweb.web.views.tableview.PyValTableView.AbstractColumn method), 234
 sortvalue() (cubicweb.web.views.tableview.RsetTableView.AbstractColumn method), 231
 sortvalue() (cubicweb.web.views.tableview.TableMixIn.AbstractColumn method), 236, 237
 SourceException, 397
 specified_etype_implements (class in cubicweb.predicates), 416
 split_rset() (cubicweb.rset.ResultSet method), 429
 StartupView (class in cubicweb.view), 212
 static_directory (cubicweb.web.webconfig.WebConfiguration attribute), 298
 static_file_add() (cubicweb.web.webconfig.WebConfiguration method), 298
 static_file_del() (cubicweb.web.webconfig.WebConfiguration method), 298
 static_file_exists() (cubicweb.web.webconfig.WebConfiguration method), 298
 static_file_open() (cubicweb.web.webconfig.WebConfiguration method), 298
 StatusResponse, 432
 StringField (class in cubicweb.web.formfields), 266
 SubmitButton (class in cubicweb.web.formwidgets), 272
 subscribe_to_debug_channel() (in module cubicweb.debug), 196
 syntax_tree() (cubicweb.rset.ResultSet method), 429
 system_sql() (cubicweb.repoapi.Connection method), 209

T

TRequest
 table_actions() (cubicweb.web.views.pyviews.PyValTableView method), 235
 table_actions() (cubicweb.web.views.tableview.TableMixIn method), 238
 table_size (cubicweb.web.views.pyviews.PyValTableView property), 235

`table_size` (*cubicweb.web.views.tableview.EntityTableView* property), 234
`table_size` (*cubicweb.web.views.tableview.RsetTableView* property), 232
`table_size` (*cubicweb.web.views.tableview.TableMixin* attribute), 237
`table_size` (*cubicweb.web.views.tableview.TableMixin* property), 238
`TableLayout` (class in *cubicweb.web.views.tableview*), 235
`TableMixin` (class in *cubicweb.web.views.tableview*), 236
`TableMixin.AbstractColumnRenderer` (class in *cubicweb.web.views.tableview*), 236, 237
`tearDown()` (*cubicweb.devtools.testlib.CubicWebTC* method), 181
`temporary_permissions()` (*cubicweb.devtools.testlib.CubicWebTC* method), 181
`TestCase` (class in *logilab.common.testlib*), 179
`TextArea` (class in *cubicweb.web.formwidgets*), 270
`TextInput` (class in *cubicweb.web.formwidgets*), 269
`TextView` (class in *cubicweb.web.views.baseviews*), 227
`TimeField` (class in *cubicweb.web.formfields*), 267
`TimeIntervalField` (class in *cubicweb.web.formfields*), 267
`title()` (*cubicweb.entities.adapters.IDublinCoreAdapter* method), 149
`traced_selection` (class in *logilab.common.registry*), 410
`transaction_actions()` (*cubicweb.repoapi.Connection* method), 209
`transaction_data` (*cubicweb.repoapi.Connection* attribute), 209
`transaction_info()` (*cubicweb.repoapi.Connection* method), 209
`transformed_rset()` (*cubicweb.rset.ResultSet* method), 429
`triples()` (*cubicweb.entities.adapters.CWUserRDFAdapter* method), 149
`triples()` (*cubicweb.entities.adapters.EntityRDFAdapter* method), 149
`TweenHandler` (class in *cubicweb.pyramid.bwcompat*), 419
`type()` (*cubicweb.entities.adapters.IDublinCoreAdapter* method), 150
`typed_value()` (*cubicweb.cwvreg.CWRegistryStore* method), 404
`typed_value()` (*cubicweb.web.formfields.Field* method), 266
`TZDatetimeField` (class in *cubicweb.web.formfields*), 267
`ucsvreader()` (in module *cubicweb.dataimport*), 411
`ucsvreader_pb()` (in module *cubicweb.dataimport*), 411
`Unauthorized`, 398
`undo_transaction()` (*cubicweb.repoapi.Connection* method), 209
`undoable_transactions()` (*cubicweb.repoapi.Connection* method), 209
`UndoTransactionException`, 399
`UniqueTogetherError`, 398
`UnknownEid`, 398
`UnknownProperty`, 398
`unregister()` (*logilab.common.registry.Registry* method), 408
`unregister()` (*logilab.common.registry.RegistryStore* method), 407
`unsubscribe_to_debug_channel()` (in module *cubicweb.debug*), 196
`update_schema()` (*cubicweb.cwvreg.CWRegistryStore* method), 404
`UpdateLoginTimeAuthenticationPolicy` (class in *cubicweb.pyramid.auth*), 418
`uri` (*cubicweb.entities.adapters.EntityRDFAdapter* attribute), 149
`url_parse_qs()` (*cubicweb.req.RequestSessionBase* method), 206, 426
`url_publish()` (*cubicweb.devtools.testlib.CubicWebTC* method), 182
`url_quote()` (*cubicweb.req.RequestSessionBase* method), 206, 426
`url_redirection_tween_factory()` (in module *cubicweb.pyramid.url_redirection*), 424
`url_unquote()` (*cubicweb.req.RequestSessionBase* method), 206, 426
`URLAttributeView` (class in *cubicweb.web.views.primary*), 220
`URLPathEvaluator` (class in *cubicweb.web.views.urlpublishing*), 430
`URLPublisherComponent` (class in *cubicweb.web.views.urlpublishing*), 242, 430
`URLRewriteEvaluator` (class in *cubicweb.web.views.urlpublishing*), 243, 431
`URLRewriter` (class in *cubicweb.web.views.urlrewrite*), 244, 431
`use_extid_as_cwuri()` (in module *cubicweb.dataimport.importer*), 192
`user_data()` (*cubicweb.req.RequestSessionBase* method), 206, 426
`user_property_keys()` (*cubicweb.cwvreg.CWRegistryStore* method), 404

V

[validation_error\(\)](#) (in module *cubicweb*), 400
[ValidationError](#), 399
[values\(\)](#) (*cubicweb.cwvreg.CWRegistryStore* method), 404
[values\(\)](#) (*cubicweb.web.formwidgets.FieldWidget* method), 268
[VerbatimAttributeView](#) (class in *cubicweb.web.views.primary*), 220
[View](#) (class in *cubicweb.view*), 211
[view\(\)](#) (*cubicweb.devtools.testlib.CubicWebTC* method), 182
[view\(\)](#) (*cubicweb.req.RequestSessionBase* method), 206, 426
[ViewsRegistry](#) (class in *cubicweb.cwvreg*), 405
[vocabulary\(\)](#) (*cubicweb.web.facet.VocabularyFacet* method), 291
[VocabularyFacet](#) (class in *cubicweb.web.facet*), 291
[vregistry: registration_callback](#), 119

W

[warning\(\)](#) (*cubicweb.appobject.AppObject* method), 402
[warning\(\)](#) (*cubicweb.repoapi.Connection* method), 210
[warning\(\)](#) (*cubicweb.web.application.CubicWebPublisher* method), 201
[write_links\(\)](#) (*cubicweb.web.views.navigation.SortedNavigation* method), 228
[wsgi_application\(\)](#) (in module *cubicweb.pyramid*), 417
[wsgi_application_from_cwconfig\(\)](#) (in module *cubicweb.pyramid*), 417

Y

[yes](#) (class in *cubicweb.appobject*), 411
[yes](#) (class in *logilab.common.registry*), 409