py4web Documentation

Release 1.20201024.1

November 05, 2020

Table of Contents

Chapter 1

What is py4web?

PY4WEB is a web framework for rapid development of efficient database driven web applications. It is an evolution of the popular web2py framework but much faster and slicker. Its internal design has been much simplified compared to web2py.

PY4WEB can be seen as a competitor of other frameworks like Django or Flask, and it can indeed serve the same purpose. Yet PY4WEB aims to provide a larger feature set out of the box and reduce the development time of new apps.

From a historical perspective our story starts in 2007 when web2py was first released. web2py was designed to provide an all-inclusive solution for web development: one zip file containing the Python interpreter, the framework, a web based IDE, and a collection of battle-tested packages that work well together. In many ways web2py has been immensely successful. Web2py succeeded in providing a low barrier of entry for new developers, a very secure development platform, and remains backwards compatible until today.

Web2py always suffered from one problem: its monolithic design. The most experienced Python developers did not understand how to use its components outside of the framework and how to use third party components within the framework. This was for good reason, as we did not care too much about them. We thought of web2py as a perfect tool that did not have to be broken into pieces because that would compromise its security. It turned out that we were wrong, and playing well with others is important. Hence, in the last two years we worked on three fronts:

- We ported web2py to Python 3.
- We broke web2py into modules that can be used independently.
- We reassembled some of those modules into a new more modular framework … PY4WEB.

PY4WEB is more than a repackaging of those modules. It is a complete redesign. It uses some of the web2py modules, but not all of them. In some cases, it uses other and better modules. Some functionality was removed and some was added. We tried to preserve most of the syntax and features that experienced web2py users loved. Here is a more explicit list:

- PY4WEB, unlike web2py, requires Python 3.
- PY4WEB, unlike web2py, can be installed using pip and its dependencies are managed using requirements.txt.
- PY4WEB apps are regular Python modules. This is very different to web2py. In particular, we ditched the custom importer, and we rely now exclusively on the regular Python import mechanism.
- PY4WEB, like web2py, can serve multiple applications concurrently, as long as the apps are submodules of the apps module.
- PY4WEB, unlike web2py, is based on bottlepy and in particular uses the Bottle request object and the

Bottle routing mechanism.

- PY4WEB, unlike web2py, does not create a new environment at every request. It introduces the concept of fixtures to explicitly declare which objects need to be re-initialized when a new http request is processed. This makes it much faster.
- PY4WEB, has a new sesson object which, like web2py's, provides strong security and encryption of the session data, but sessions are no longer stored in the file system - which created performance issues. It provides sessions in cookies, in redis, in memcache, or in database. We also limited session data to objects that are json serializable.
- PY4WEB, like web2py, has a built-in ticketing system but, unlike web2py, this system is global and not per app. Tickets are no longer stored in the filesystem with the individual apps. They are stored in a single database.
- PY4WEB, like web2py, is based on pydal but uses some new features of pydal (RESTAPI).
- PY4WEB, like web2py, uses the yatl template language but defaults to square brackets delimiters to avoid conflicts with model JS frameworks, such as Vue.js and angular.js. Yatl includes a subset of the web2py helpers.
- PY4WEB, unlike web2py, uses the pluralization library for internationalization. In practice, this exposes an object T very similar to web2py's T but it provides better caching and more flexible pluralization capabilities.
- PY4WEB comes with a Dashboard APP that replaces web2py's admin. This is a web IDE for uploading/managing/editing apps.
- PY4WEB's Dashboard includes a web based database interface. This replaces the appadmin functionality of web2py.
- PY4WEB comes with a Form object that is similar to web2py's SQLFORM but it is much simpler and faster. The syntax is the same. This has been provided in order to help users port existing apps; but PY4WEB encourages using API based forms over postbacks.
- PY4WEB comes with an Auth object that replaces the web2py one. It is more modular and easier to extend. Out of the box, it provides the basic functionality of register, login, logout, change password, request change password, edit profile as well as integration with PAM, SAML2, LDAP, OAUTH2 (google, facebook, and twitter).
- PY4WEB comes with some utilites like "tags", for instance, which allows adding searchable tags to any database table. It can be used, for example, to tag users with groups and search users by groups and apply permissions based on membership.
- PY4WEB comes with with some custom Vue.js components designed to interact with the PyDAL RESTAPI, and with PY4WEB in general. These APIs are designed to allow the server to set policies about which operations a client is allowed to perform, but give the client flexibility within those constraints. The two main components are mtable (which provides a web based interface to the database similar to web2py's grid) and auth (a customizable interface to the Auth API).

The goal of PY4WEB is and remains the same as web2py's: to make web development easy and accessible, while producing applications that are fast and secure.

Chapter 2

Installation and Startup

2.1 Supported platforms and prerequisites

PY4WEB runs fine on Windows, MacOS and Linux. Its only prerequisite is Python 3.6+, which must be installed in advance (except if you use binaries).

2.2 Setup procedures

There are four alternative ways of running py4web, with different level of difficulty and flexibility. Let's look at the pros and cons.

2.2.1 Installing from binaries

This is not a real installation, because you just copy a bunch of files on your system without modifying it anyhow. Hence this is the simplest solution, especially for newbies or students, because it does not require Python pre-installed on your system nor administrative rights. On the other hand, it's experimental, it could contain an old py4web release and it is quite difficult to add other functionalities to it.

In order to use it you just need to download the latest Windows or MacOS ZIP file from this external repository. Unzip it on a local folder and open a command line there. Finally run

```
py4web-start set_password
py4web-start run apps
```

With this type of installation, remember to always use **py4web-start** instead of 'py4web' or 'py4web.py' in the following documentation.

2.2.2 Hint: use a virtual environment (virtualenv)

A full installation of any complex python application like py4web will surely modify the python environment of your system. In order to prevent any unwanted change, it's a good habit to use a python virtual environment (also called **virtualenv**, see here for an introduction). This is a standard python feature; if you still don't know virtualenv it's a good time to start its discovery!

Activate it before using any of the following *real* installation procedures is highly reccomended.

2.2.3 Installing from pip

Using *pip* is the standard installation procedure for py4web. From the command line

```
python3 -m pip install --upgrade py4web --no-cache-dir --user
```

but do **not** type the *–user* option with virtualenv or a standard Windows installation which is already per-user.

Also, if *python3* does not work, try with the simple *python* command instead.

This will install py4web and all its dependencies on the system's path only. The assets folder (that contains the py4web's system apps) will also be created. After the installation you'll be able to start py4web on any given working folder with

```
py4web setup apps
py4web set_password
py4web run apps
```

If the command py4web is not accepted, it means it's not in the system's path. On Windows, a special py4web.exe file (pointing to py4web.py) will be created by *pip* on the system's path, but not if you type the *–user* option by mistake.

2.2.4 Installing from source (globally)

This is the traditional way for installing a program, but it works only on Linux and MacOS. All the requirements will be installed on the system's path along with links to the py4web.py program on the local folder

```
git clone https://github.com/web2py/py4web.git
cd py4web
make assets
make test
make install
py4web run apps
```

Also notice that when installing in this way the content of py4web/assets folder is missing at first but it is manually created later with the 'make assets' command.

2.2.5 Installing from source (locally)

In this way all the requirements will be installed or upgraded on the system's path, but py4web itself will only be copied
on a local folder. This is especially useful if you already have a working py4web installation but you want to test a different
one. From the command line, go to a given working folder and then run

```
git clone https://github.com/web2py/py4web.git
cd py4web
python3 -m pip install  --upgrade -r requirements.txt
```

Once installed, you should always start it from there with

For Linux / MacOS

```
./py4web.py setup apps
./py4web.py set_password
```

```
./py4web.py run apps
```

If you have installed py4web both globally and locally, notice the **./** ; it forces the run of the local folder's py4web and not the globally installed one.

For Windows

```
python3 py4web.py setup apps
python3 py4web.py set_password
python3 py4web.py run apps
```

On Windows, the programs on the local folder are always executed before the ones in the path (hence you don't need the **./**). But running .py files directly it's not usual and you'll need an explicit python3/python command.

2.3 Upgrading

If you installed py4web from pip you can simple upgrade it with

```
python3 -m pip install --upgrade py4web
```

Mind that it will not automatically upgrade the apps like **Dashboard** and **Default**. You have to manually remove these apps and then run

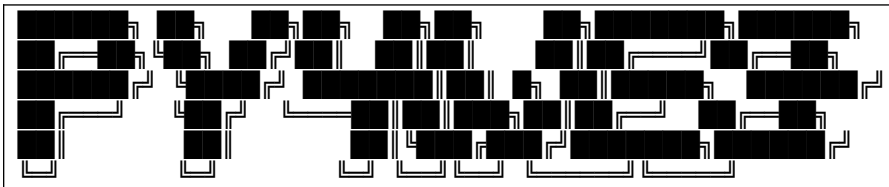
```
py4web setup apps
```

in order to re-install them. This is a safety precaution, in case you made changes to those apps.

If you installed py4web in any other way, you must upgrade it manually. First you have to make a backup of any personal py4web work you've done, then delete the old installation folder and re-install the framework again.

2.4 First run

Running py4web using any of the previous procedure should produce an output like this



```
Dashboard is at: http://127.0.0.1:8000/_dashboard
[X] loaded _dashboard
[X] loaded _default
Bottle v0.12.16 server starting up (using TornadoServer())...
Listening on http://127.0.0.1:8000/
Hit Ctrl-C to quit.
```

Generally `apps` is the name of the folder where you keep all your apps, and can be explicitly set wit the _'run_' command. If that folder does not exist, it is created. PY4WEB expects to find at least two apps in this folder: **Dashboard** (_dashboard) and **Default** (_default). If it does not find them, it installs them.

Dashboard is a web based IDE.

Default is an app that does nothing other than welcome the user.

Notice that some apps - like **Dashboard** and **Default** - have a special role in py4web and therefore their actual name starts with _ to avoid conflicts with apps created by you.

Once py4web is running you can access a specific app at the following urls:

```
http://localhost:8000
http://localhost:8000/_dashboard
http://localhost:8000/{yourappname}/index
```

Notice that ONLY the **Default** app is special because if does not require the "{appname}/" prefix in the path, like all the other apps do. In general you you may want to symlink `apps/_default` to your default app.

For all apps the trailing `/index` is optional.

2.5 Command line options

py4web provides multiple command line options which can be listed by running it with the **–help** argument

```
# py4web --help
Usage: py4web.py [OPTIONS] COMMAND [ARGS]...

  PY4WEB - a web framework for rapid development of efficient database
  driven web applications

  Type "py4web COMMAND -h" for available options on commands

Options:
  -help, -h, --help  Show this message and exit.

Commands:
  call          Call a function inside apps_folder
  run           Run all the applications on apps_folder
  set_password  Set administrator's password for the Dashboard
  setup         Setup new apps folder or reinstall it
  shell         Open a python shell with apps_folder added to the path
  version       Show versions and exit
```

2.5.1 call command option

```
# py4web call -h
Usage: py4web.py call [OPTIONS] APPS_FOLDER FUNC

  Call a function inside apps_folder

Options:
  --args TEXT        Arguments passed to the program/function  [default: {}]
  -help, -h, --help  Show this message and exit.
```

2.5.2 run command option

```
# py4web run -h
```

```
Usage: py4web.py run [OPTIONS] [APPS_FOLDER]

  Run all the applications on apps_folder

Options:
  -Y, --yes                     No prompt, assume yes to questions  [default:
                                False]

  -H, --host TEXT               Host name  [default: 127.0.0.1]
  -P, --port INTEGER            Port number  [default: 8000]
  -p, --password_file TEXT      File for the encrypted password  [default:
                                password.txt]

  -w, --number_workers INTEGER  Number of workers  [default: 0]
  -d, --dashboard_mode TEXT     Dashboard mode: demo, readonly, full
                                (default), none  [default: full]

  --watch [off|sync|lazy]       Watch python changes and reload apps
                                automatically, modes: off (default), sync,
                                lazy
  --ssl_cert PATH               SSL certificate file for HTTPS
  --ssl_key PATH                SSL key file for HTTPS
  -help, -h, --help             Show this message and exit.
```

If you want py4web to automatically reload an application upon any changes to files of that application, you can:

- for immediate reloading (sync-mode): `py4web run --watch=sync`
- for reloading on any first incoming request to the application has been changed (lazy-mode): `py4web run --watch=lazy`

2.5.3 set_password command option

```
# py4web set_password -h
Usage: py4web.py set_password [OPTIONS]

  Set administrator's password for the Dashboard

Options:
  --password TEXT           Password value (asked if missing)
  -p, --password_file TEXT  File for the encrypted password  [default:
                            password.txt]

  -h, -help, --help         Show this message and exit.
```

If the `--dashboard_mode` is not `demo` or `none`, every time py4web starts, it asks for a one-time password for you to access the dashboard. This is annoying. You can avoid it by storing a pdkdf2 hashed password in a file (by default called password.txt) with the command

```
py4web set_password
```

It will not ask again unless the file is deleted. You can also use a custom file name with

```
py4web set_password my_password_file.txt
```

and then ask py4web to re-use that password at runtime with

```
py4webt run -p my_password_file.txt apps
```

Finally you can manually create the file yourself with:

```
$ python3 -c "from pydal.validators import CRYPT;
open('password.txt','w').write(str(CRYPT()(input('password:'))[0]))"
password: *****
```

2.5.4 setup command option

```
# py4web setup -h
Usage: py4web.py setup [OPTIONS] [APPS_FOLDER]

  Setup new apps folder or reinstall it

Options:
  -Y, --yes           No prompt, assume yes to questions  [default: False]
  -help, -h, --help   Show this message and exit.
```

This option create a new apps folder (or reinstall it). If needed, it will ask for the confirmation of the new folder's creation and then for copying every standard py4web apps from the assets folder. It currently does nothing on binaries installations and from source installation (locally) - for them you can manually copy the existing apps folder to the new one.

2.5.5 shell command option

```
# py4web shell -h
Usage: py4web.py shell [OPTIONS] [APPS_FOLDER]

  Open a python shell with apps_folder added to the path

Options:
  -h, -help, --help   Show this message and exit.
```

Py4web's shell is just the regular python shell with apps added to the search path. Notice that the shell is for all the apps, not a single one. You can then import the needed modules from the apps you need to access.

For example, inside a shell you can

```
from apps.myapp import db
from py4web import Session, Cache, Translator, DAL, Field
from py4web.utils.auth import Auth
```

2.5.6 version command option

```
# py4web version -h
Usage: py4web.py version [OPTIONS]

  Show versions and exit

Options:
  -a, --all           List version of all modules
  -h, -help, --help   Show this message and exit.
```

With the *-a* option you'll get the version of all the available python modules, too.

2.6 Deployment on the cloud

2.6.1 Deployment on GCloud (aka Google App Engine)

Login into the Gcloud console (https://console.cloud.google.com/) and create a new project. You will obtain a project id that looks like "{project_name}-{number}".

In your local file system make a new working folder and cd into it:

```
mkdir gae
cd gae
```

Copy the example files from py4web (assuming you have the source from github)

```
cp /path/to/py4web/development_tools/gcloud/* ./
```

Copy or symlink your `apps` folder into the gae folder, or maybe make a new apps folder containing an empty `__init__.py` and symlink the individual apps you want to deploy. You should see the following files/folders:

```
Makefile
apps
  __init__.py
  ... your apps ...
lib
app.yaml
main.py
```

Install the Google SDK, py4web and setup the working folder:

```
make install-gcloud-linux
make setup
gcloud config set {your email}
gcloud config set {project id}
```

(replace {your email} with your google email account and {project id} with the project id obtained from Google).

Now every time you want to deploy your apps, simply do:

```
make deploy
```

You may want to customize the Makefile and app.yaml to suit your needs. You should not need to edit `main.py`.

2.6.2 Deployment on PythonAnywhere.com

Watch the video: https://youtu.be/Wxjl_vkLAEY and follow the detailed tutorial on https://github.com/tomcam/py4webcasts/blob/master/docs/how-install-source-pythonanywhere.md . The bottle_app.py script is in `py4web/deployment_tools/pythonanywhere.com/bottle_app.py`

Chapter 3

Creating your first app

3.1 From scratch

Apps can be created using the dashboard or directly from the filesystem. Here, we are going to do it manually, as the Dashboard is described in its own chapter.

Keep in mind that an app is a Python module; therefore it needs only a folder and a `__init__.py` file in that folder:

```
mkdir apps/myapp
echo '' > apps/myapp/__init__.py
```

Notice that for Windows, you must use backslashes (i.e. '') instead of slashes. Also, an empty **init**.py file is not strictly needed since Python 3.3, but it will be useful later on. If you now restart py4web or press the "Reload Apps" in the Dashboard, py4web will find this module, import it, and recognize it as an app, simply because of its location. An app is not required to do anything. It could just be a container for static files or arbitrary code that other apps may want to import and access. Yet typically most apps are designed to expose static or dynamic web pages.

3.2 Static web pages

To expose static web pages you simply need to create a `static` subfolder, and any file in there will be automatically published:

```
mkdir apps/myapp/static
echo 'Hello World' > apps/myapp/static/hello.txt
```

The newly created file will be accessible at

```
http://localhost:8000/myapp/static/hello.txt
```

Notice that `static` is a special path for py4web and only files under the `static` folder are served.

Important: internally py4web uses the bottle static_file method for serving static files, which means it supports streaming, partial content, range requests, and if-modified-since. This is all handled automatically based on the http request headers.

3.3 Dynamic Web Pages

To create a dynamic page, you must create a function that returns the page content. For example edit the `myapp/__init__.py` as follows:

```
import datetime
from py4web import action

@action('index')
def page():
    return "hello, now is %s" % datetime.datetime.now()
```

Restart py4web or press the Dashboard "Reload Apps" button, and this page will be accessible at

```
http://localhost:8000/myapp/index
```

or

```
http://localhost:8000/myapp
```

(notice that index is optional)

Unlike other frameworks, we do not import or start the webserver within the `myapp` code. This is because py4web is already running, and it may be serving multiple apps. py4web imports our code and exposes functions decorated with `@action()`. Also notice that py4web prepends `/myapp` (i.e. the name of the app) to the url path declared in the action. This is because there are multiple apps, and they may define conflicting routes. Prepending the name of the app removes the ambiguity. But there is one exception: if you call your app `_default`, or if you create a symlink from `_default` to `myapp`, then py4web will not prepend any prefix to the routes defined inside the app.

3.3.1 On return values

py4web actions should return a string or a dictionary. If they return a dictionary you must tell py4web what to do with it. By default py4web will serialize it into json. For example edit `__init__.py` again and add

```
@action('colors')
def colors():
    return {'colors': ['red', 'blue', 'green']}
```

This page will be visible at

```
http://localhost:8000/myapp/colors
```

and returns a JSON object `{"colors": ["red", "blue", "green"]}`. Notice we chose to name the function the same as the route. This is not required, but it is a convention that we will often follow.

You can use any template language to turn your data into a string. PY4WEB comes with yatl, a full chapter will be dedicated later and we will provide an example shortly.

3.3.2 Routes

It is possible to map patterns in the URL into arguments of the function. For example:

```
@action('color/<name>')
```

```
def color(name):
    if name in ['red', 'blue', 'green']:
        return 'You picked color %s' % name
    return 'Unknown color %s' % name
```

This page will be visible at

```
http://localhost:8000/myapp/color/red
```

The syntax of the patterns is the same as the Bottle routes. A route wildcard can be defined as

- `<name>` or
- `<name:filter>` or
- ``````

And these are possible filters (only `:` has a config):

- `:int` matches (signed) digits and converts the value to integer.
- `:float` similar to :int but for decimal numbers.
- `:path` matches all characters including the slash character in a non-greedy way, and may be used to match more than one path segment.
- ``:re[:exp]``` allows you to specify a custom regular expression in the config field. The matched value is not modified.

The pattern matching the wildcard is passed to the function under the specified variable `name`.

Also, the action decorator takes an optional `method` argument that can be an HTTP method or a list of methods:

```
@action('index', method=['GET','POST','DELETE'])
```

You can use multiple decorators to expose the same function under multiple routes.

3.3.3 The `request` object

From py4web you can import `request`

````python from py4web import request

@action('paint') def paint(): if 'color' in request.query return 'Painting in %s' % request.query.get('color') return 'You did not specify a color'

```
This action can be accessed at:
```

http://localhost:8000/myapp/paint?color=red

```
Notice that the request object is the a [Bottle request
object](https://bottlepy.org/docs/dev/api.html#the-request-object)

#### Templates

In order to use a yatl template you must declare it. For example create a file
```apps/myapp/templates/paint.html``` that contains:

```html
<body>
```
```

```
  <head>
    <style>
      body {background}
    </style>
  </head>
  <body>
    <h1>Color</h1>
 </body>
</html>
```

then modify the paint action to use the template and default to green.

```
@action('paint')
@action.uses('paint.html')
def paint():
    return dict(color = request.query.get('color', 'green'))
```

The page will now display the color name on a background of the corresponding color.

The key ingredient here is the decorator `@action.uses(...)`. The arguments of `action.uses` are called **fixtures**. You can specify multiple fixtures in one decorator or you can have multiple decorators. Fixtures are objects that modify the behavior of the action, that may need to be initialized per request, that may filter input and output of the action, and that may depend on each-other (they are similar in scope to Bottle plugins but they are declared per-action, and they have a dependency tree which will be explained later).

The simplest type of fixture is a template. You specify it by simply giving the name of the file to be used as template. That file must follow the yatl syntax and must be located in the `templates` folder of the app. The object returned by the action will be processed by the template and turned into a string.

You can easily define fixtures for other template languages. This is described later.

Some built-in fixtures are:

- the DAL object (which tells py4web to obtain a database connection from the pool at every request, and commit on success or rollback on failure)
- the Session object (which tells py4web to parse the cookie and retrieve a session at every request, and to save it if changed)
- the Translator object (which tells py4web to process the accept-language header and determine optimal internationalization/pluralization rules)
- the Auth object (which tells py4web that the app needs access to the user info)

They may depend on each other. For example, the Session may need the DAL (database connection), and Auth may need both. Dependencies are handled automatically.

## 3.4 From _scaffold

Most of the times, you do not want to start writing code from scratch. You also want to follow some sane conventions outlined here, like not putting all your code into `__init__.py`. PY4WEB provides a Scaffolding (_scaffold) app, where files are organized properly and many useful objects are pre-defined.

You will normally find the scaffold app under apps, but you can easily create a new clone of it manually or using the Dashboard.

Here is the tree structure of the `_scaffold` app:

```
├── __init__.py           # imports everything else
├── common.py             # defines useful objects
├── controllers.py        # your actions
├── databases             # your sqlite databases and metadata
│   └── README.md
├── models.py             # your pyDAL table model
├── settings.py           # any settings used by the app
├── settings_private.py   # (optional) settings that you want to keep private
├── static                # static files
│   ├── README.md
│   ├── components        # py4web's vue auth component
│   │   ├── auth.html
│   │   └── auth.js
│   ├── css               # CSS files, we ship bulma because it is JS agnostic
│   │   └── no.css        # we used bulma.css in the past
│   ├── favicon.ico
│   └── js                # JS files, we ship with these but you can replace them
│       ├── axios.min.js
│       ├── sugar.min.js
│       ├── utils.js
│       └── vue.min.js
├── tasks.py
├── templates             # your templates go here
│   ├── README.md
│   ├── auth.html         # the auth page for register/logic/etc (uses vue)
│   ├── generic.html      # a general purpose template
│   ├── index.html
│   └── layout.html       # a bulma layout example
└── translations          # internationalization/pluralization files go here
    └── it.json           # py4web internationalization/pluralization files are in
JSON, this is an italian example
```

The scaffold app contains an example of a more complex action:

```
from py4web import action, request, response, abort, redirect, URL
from yatl.helpers import A
from . common import db, session, T, cache, auth


@action('welcome', method='GET')
@action.uses('generic.html', session, db, T, auth.user)
def index():
    user = auth.get_user()
    message = T('Hello {first_name}'.format(**user))
    return dict(message=message, user=user)
```

Notice the following:

- `request`, `response`, `abort` are defined by Bottle
- `redirect` and `URL` are similar to their web2py counterparts
- helpers (`A`, `DIV`, `SPAN`, `IMG`, etc) must be imported from `yatl.helpers` . They work pretty much as in web2py
- `db`, `session`, `T`, `cache`, `auth` are Fixtures. They must be defined in `common.py`.
- `@action.uses(auth.user)` indicates that this action expects a valid logged-in user retrievable by `auth.get_user()`. If that is not the case, this action redirects to the login page (defined also in

`common.py` and using the Vue.js auth.html component).

When you start from scaffold, you may want to edit `settings.py`, `templates`, `models.py` and `controllers.py` but probably you don't need to change anything in `common.py`.

In your html, you can use any JS library that you want because py4web is agnostic to your choice of JS and CSS, but with some exceptions. The `auth.html` which handles registration/login/etc. uses a vue.js component. Hence if you want to use that, you should not remove it.

## 3.5 App Watchdog

Py4web facilitates a development server's setup that automatically reloads an app when its Python source files change. Any other files inside an app can be watched by setting a handler function using ``**@app_watch_handler**`` decorator.

```
--watch [off|sync|lazy]       Watch python changes and reload apps
                               automatically, modes: off (default), sync,
                               lazy
```

Two examples of its usage are reported now. Do not worry if you don't fully undestand them: the key point here is that even non-python code could be reloaded automatically if you explicit it with the ``**@app_watch_handler**`` decorator.

Watch SASS files and compile them when edited:

```
from py4web.core import app_watch_handler
import sass # https://github.com/sass/libsass-python

@app_watch_handler(
    ["static_dev/sass/all.sass",
     "static_dev/sass/main.sass",
     "static_dev/sass/overrides.sass"])
def sass_compile(changed_files):
    print(changed_files) # for info, files that changed, from a list of watched files
 above
    ## ...
    compiled_css = sass.compile(filename=filep, include_paths=includes,
output_style="compressed")
    dest = os.path.join(app, "static/css/all.css")
    with open(dest, "w") as file:
        file.write(compiled)
```

Validate javascript syntax when edited:

```
import esprima # Python implementation of Esprima from Node.js

@app_watch_handler(
    ["static/js/index.js",
     "static/js/utils.js",
     "static/js/dbadmin.js"])
def validate_js(changed_files):
    for cf in changed_files:
        print("JS syntax validation: ", cf)
        with open(os.path.abspath(cf)) as code:
            esprima.parseModule(code.read())
```

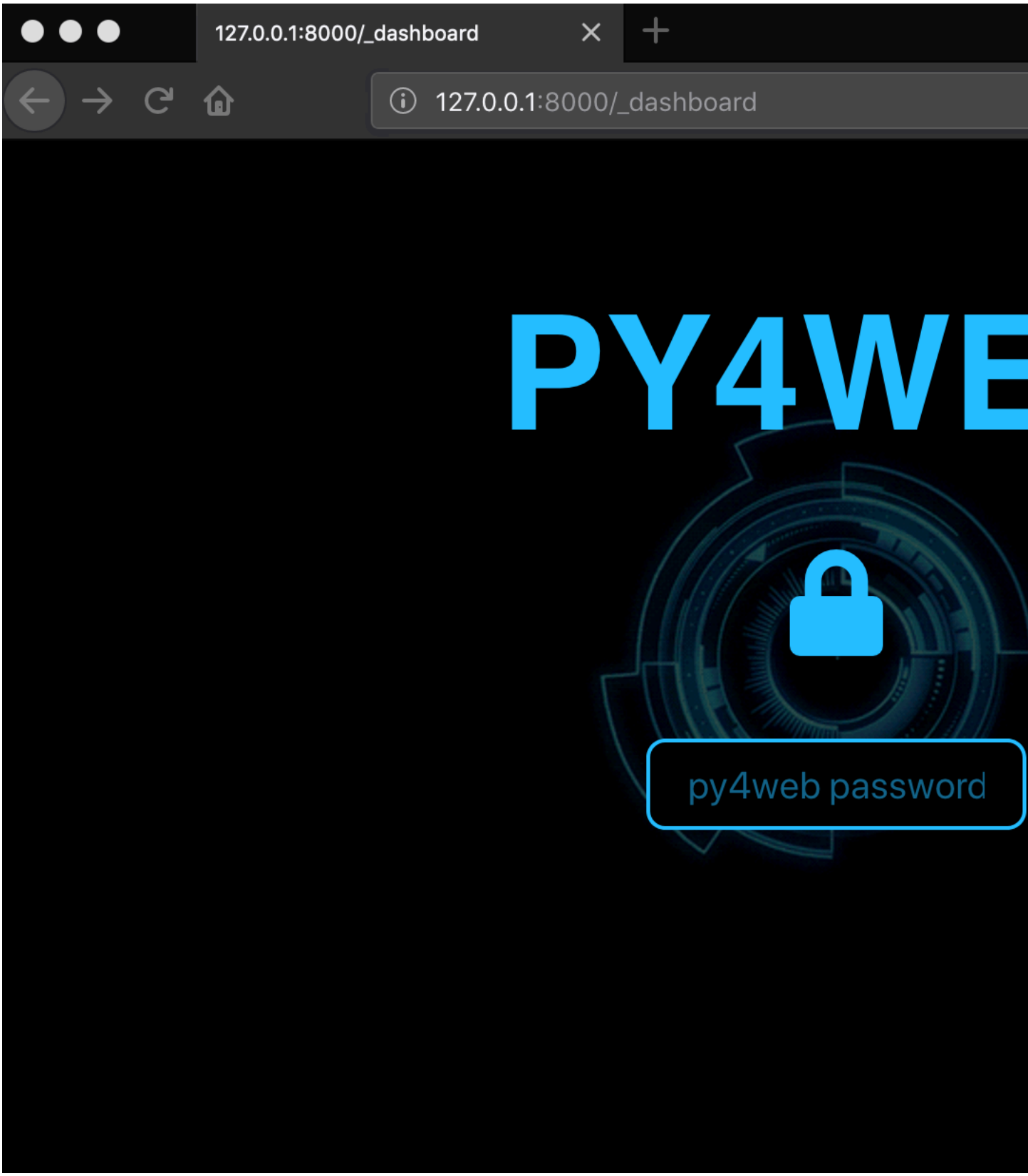
Filepaths passed to ``**@app_watch_handler**`` decorator must be relative to an app. Python files("*.py") in

a list passed to the decorator are ignored since they are watched by default. Handler function's parameter is a list of filepaths that were changed. All exceptions inside handlers are printed in terminal.

Chapter 4

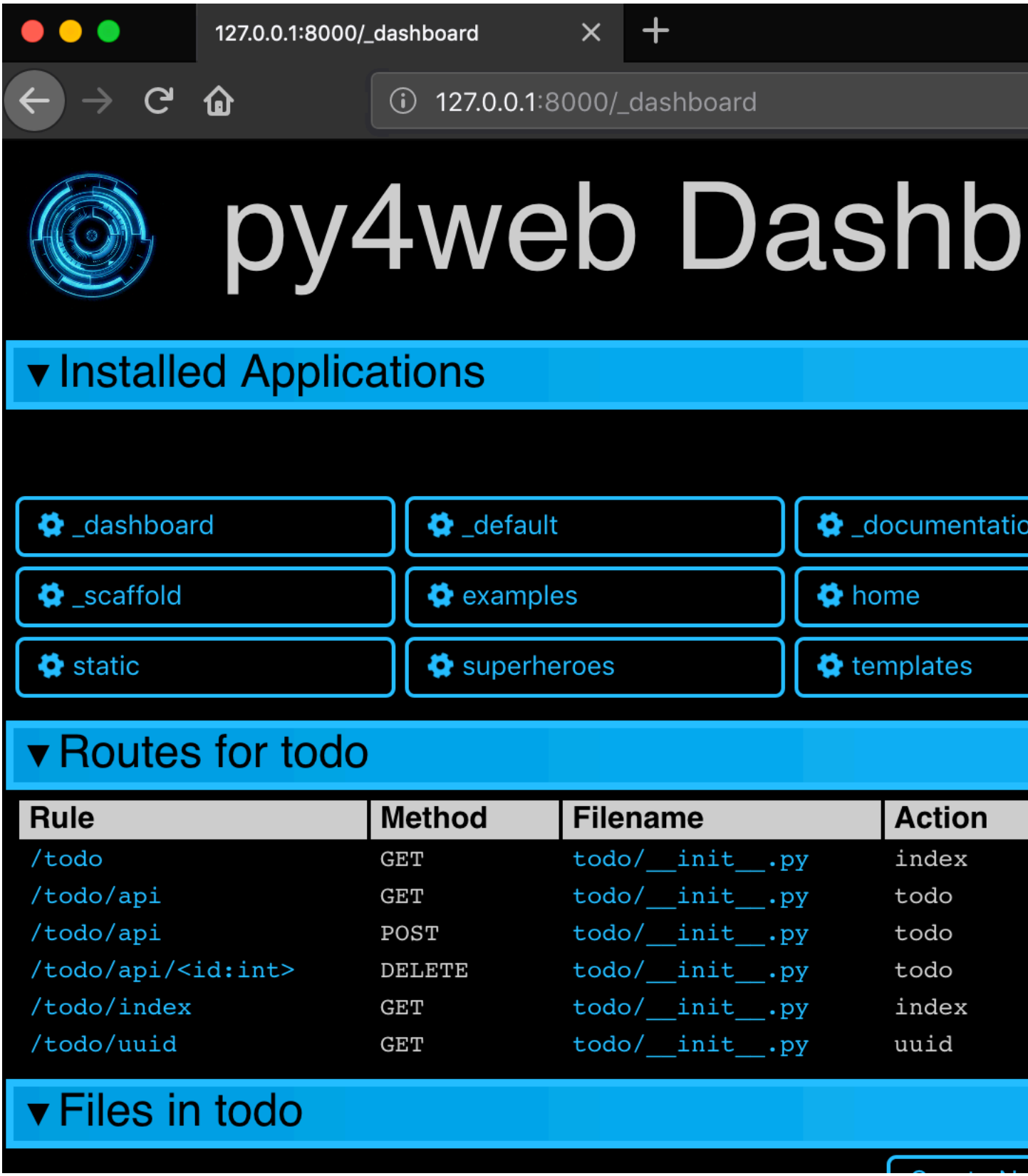
# Dashboard

Login into the dashboard

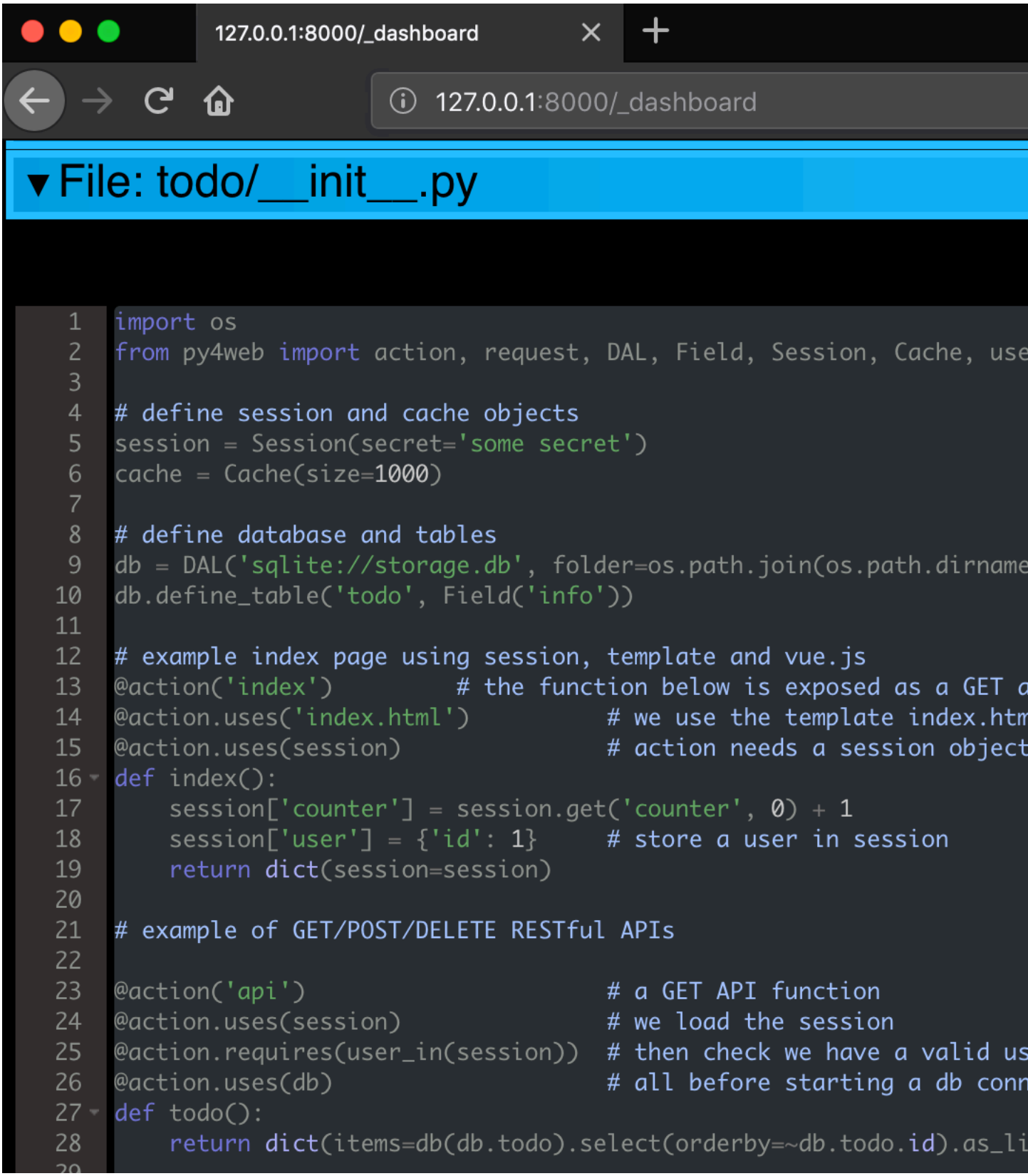



Click on a tab title to expand. Tabs are context dependent. For example, open tab "Installed Applications"

and click on an installed application to select it. This will create new tabs "Routes", "Files", and "Model" for the selected app.

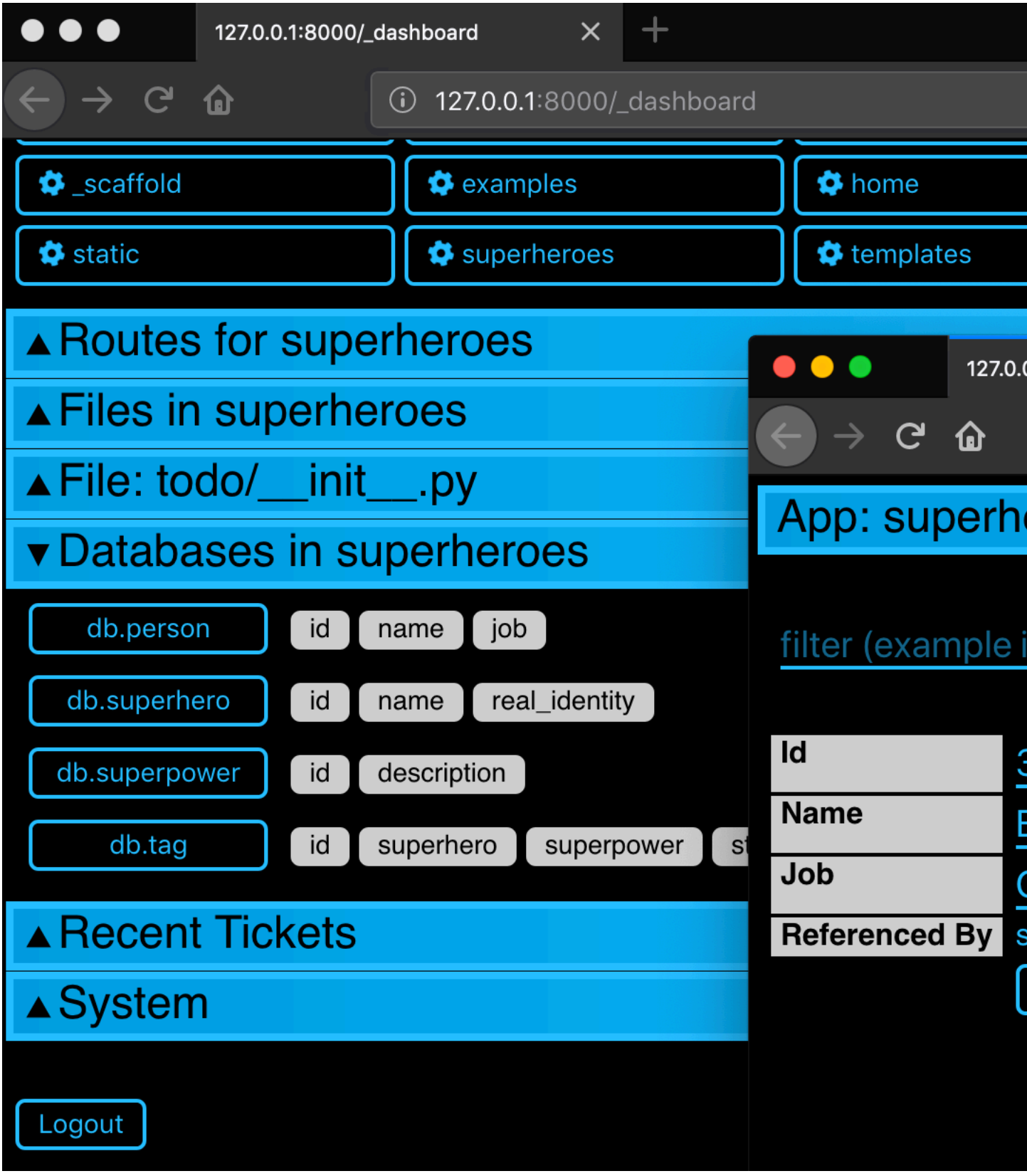


The “Files” tab allows you to browse the folder that contains the selected app and edit any file that comprises the app. If you edit a file you must click on “Reload Apps” under the “Installed Applications” tab for the change to take effect. If an app fails to load, its corresponding button is displayed in red. Click on it to see the corresponding error.

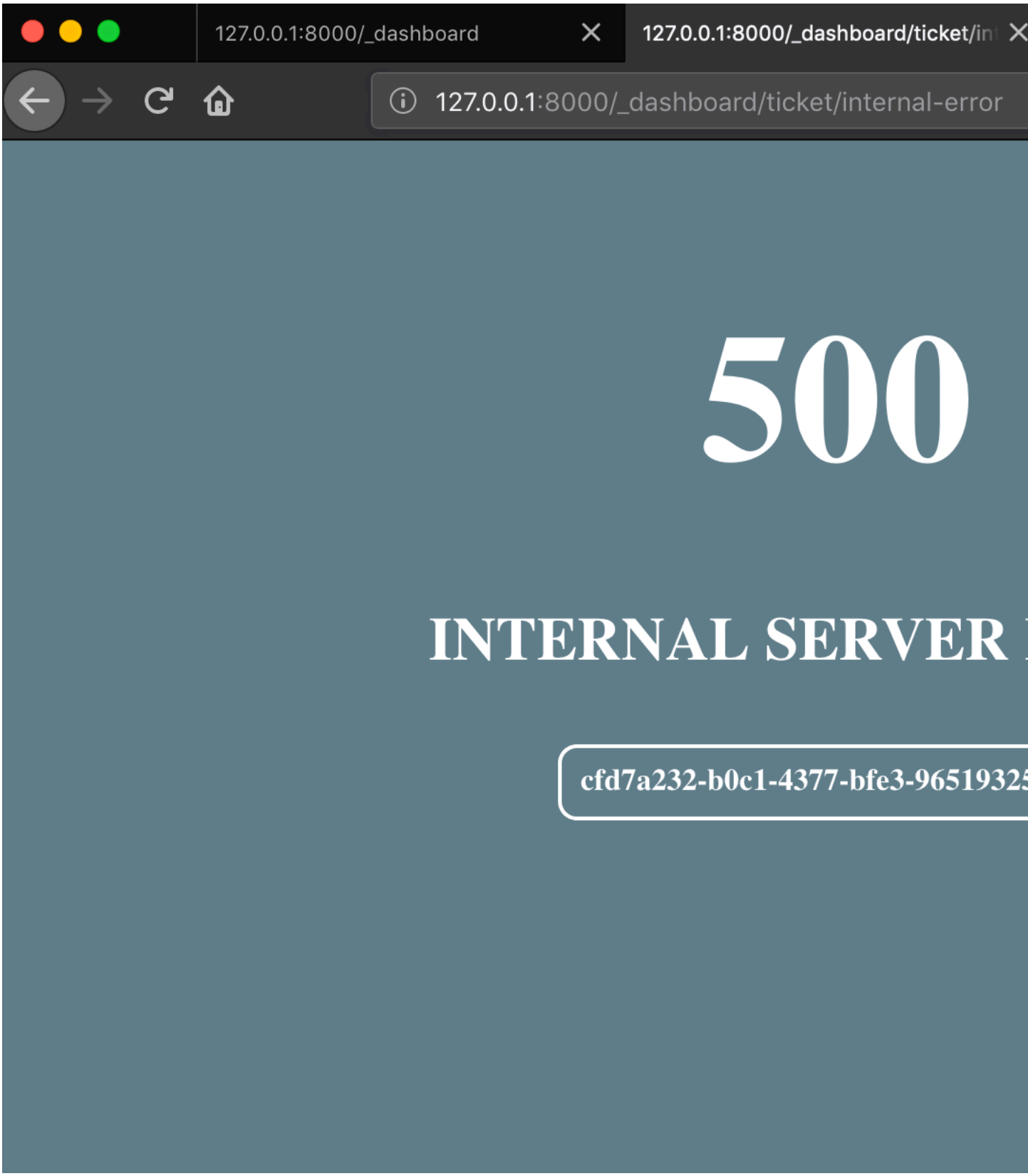



The Dashboard exposes the db of all the apps using pydal RESTAPI. It also provides a web interface to

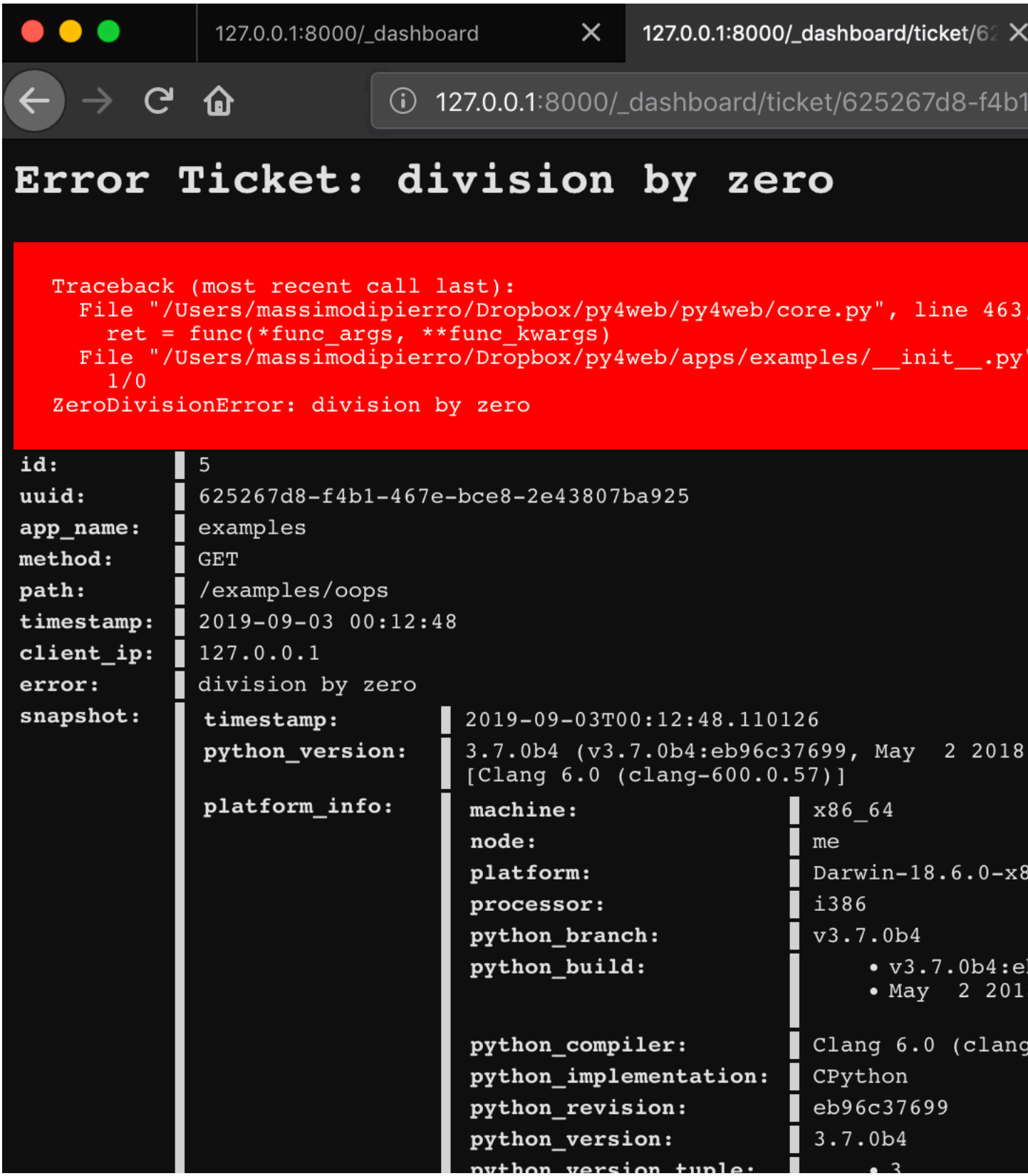
perform search and CRUD operations.



If a user visits and app and triggers a bug, the user it issued a ticket.



The ticket is logged in py4web database. The Dashboard displays the most common recent issues and allows searching tickets.



Chapter 5

# Fixtures

A fixture is defined as "a piece of equipment or furniture which is fixed in position in a building or vehicle". In our case a fixture is something attached to the action that processes an HTTP request in order to produce a response.

When processing any HTTP requests there are some optional operations we may want to perform. For example parse the cookie to look for session information, commit a database transaction, determine the preferred language from the HTTP header and lookup proper internationalization, etc. These operations are optional. Some actions need them and some actions do not. They may also depend on each other. For example, if sessions are stored in the database and our action needs it, we may need to parse the session cookie from the header, pick up a connection from the database connection pool, and - after the action has been executed - save the session back in the database if data has changed.

PY4WEB fixtures provide a mechanism to specify what an action needs so that py4web can accomplish the required tasks (and skip non required ones) in the most efficient manner. Fixtures make the code efficient and reduce the need for boilerplate code.

PY4WEB fixtures are similar to WSGI middleware and BottlePy plugin except that they apply to individual actions, not to all of them, and can depend on each other.

PY4WEB comes with some pre-defined fixtures for actions that need sessions, database connections, internationalization, authentication, and templates. Their usage will be explained in this chapter. The Developer is also free to add fixtures, for example, to handle a third party template language or third party session logic.

## 5.1 Important about Fixtures

In the examples below we will explain how to apply individual fixtures. In practice fixtures can be applied in groups. For example:

```
preferred = action.uses(Session, Auth, T, Flash)
```

Then you can apply all of the at once with:

```
@action('index.html')
@preferred
def index():
    return dict()
```

## 5.2 Templates

PY4WEB, by default uses the yatl template language and provides a fixture for it.

```
from py4web import action
from py4web.core import Template

@action('index')
@action.uses(Template('index.html', delimiters='[[ ]]'))
def index():
    return dict(message="Hello world")
```

Note: This example assumes that you created the application from the scaffolding app, so that the template index.html is already created for you.

The Template object is a Fixture. It transforms the `dict()` returned by the action into a string by using the `index.html` template file. In a later chapter we will provide an example of how to define a custom fixture to use a different template language, for example Jinja2.

Notice that since the use of templates is very common and since, most likely, every action uses a different template, we provide some syntactic sugar, and the two following lines are equivalent:

```
@action.uses('index.html')
@action.uses(Template('index.html', delimiters='[[ ]]')
```

Notice that py4web template files are cached in RAM. The py4web caching object is described later.

## 5.3 Sessions

The session object is also a Fixture. Here is a typical example of usage to implement a counter.

```
from py4web import Session, action
session = Session(secret='my secret key')

@action('index')
@action.uses(session)
def index():
    counter = session.get('counter', -1)
    counter += 1
    session['counter'] = counter
    return "counter = %i" % counter
```

Notice that the session object has the same interface as a Python dictionary.

By default the session object is stored in a cookie called, signed and encrypted, using the provided secret. If the secret changes existing sessions are invalidated. If the user switches from HTTP to HTTPS or vice versa, the user session is invalidated. Session in cookies have a small size limit (4Kbytes after being serialized and encrypted) so do not put too much into them.

In py4web sessions are dictionaries but they are stored using JSON (JWT specifically) therefore you should only store objects that are JSON serializable. If the object is not JSON serializable, it will be serialized using the `__str__` operator and some information may be lost.

By default py4web sessions never expire (unless they contain login information, but that is another story) even if an expiration can be set. Other parameters can be specified as well:

```
session = Session(secret='my secret key',
                  expiration=3600,
                  algorithm='HS256',
                  storage=None,
                  same_site='Lax')
```

- Here `algorithm` is the algorithm to be used for the JWT token signature.
- `storage` is a parameter that allows to specify an alternate session storage method (for example redis, or database).
- `same_site` is an option that prevents CSRF attacks and is enabled by default. You can read more about it here.

### 5.3.1 Session in memcache

```
import memcache, time
conn = memcache.Client(['127.0.0.1:11211'], debug=0)
session = Session(storage=conn)
```

Notice that a secret is not required when storing cookies in memcache because in this case the cookie only contains the UUID of the session.

### 5.3.2 Session in redis

```
import redis
conn = redis.Redis(host='localhost', port=6379)
conn.set = lambda k, v, e, cs=conn.set, ct=conn.ttl: (cs(k, v), e and ct(e))
session = Session(storage=conn)
```

Notice: a storage object must have `get` and `set` methods and the `set` method must allow to specify an expiration. The redis connection object has a `ttl` method to specify the expiration, hence we monkey patch the `set` method to have the expected signature and functionality.

### 5.3.3 Session in database

```
from py4web import Session, DAL
from py4web.utils.dbstore import DBStore
db = DAL('sqlite:memory')
session =  Session(storage=DBStore(db))
```

A secret is not required when storing cookies in the database because in this case the cookie only contains the UUID of the session.

Also this is one case when the a fixture (session) requires another fixture (db). This is handled automatically by py4web and the following are equivalent:

```
@action.uses(session)
@action.uses(db, session)
```

### 5.3.4 Session anywhere

You can easily store sessions in any place you want. All you need to do is provide to the `Session` object a `storage` object with both `get` and `set` methods. For example, imagine you want to store sessions on your local filesystem:

```
import os
import json

class FSStorage:
    def __init__(self, folder):
        self.folder = folder
    def get(self, key):
        filename = os.path.join(self.folder, key)
        if os.path.exists(filename):
            with open(filename) as fp:
                return json.load(fp)
        return None
    def set(self, key, value, expiration=None):
        filename = os.path.join(self.folder, key)
        with open(filename, 'w') as fp:
            json.dump(value, fp)

session = Session(storage=FSStorage('/tmp/sessions'))
```

We leave to you as an exercise to implement expiration, limit the number of files per folder by using subfolders, and implement file locking. Yet we do not recomment storing sessions on the filesystem: it is inefficient and does not scale well.

## 5.4 Translator

Here is an example of usage:

```
from py4web import action, Translator
import os

T_FOLDER = os.path.join(os.path.dirname(__file__), 'translations')
T = Translator(T_FOLDER)

@action('index')
@action.uses(T)
def index(): return str(T('Hello world'))
```

The string 'hello world` will be translated based on the internationalization file in the specified "translations" folder that best matches the HTTP `accept-language` header.

Here `Translator` is a py4web class that extends `pluralize.Translator` and also implements the `Fixture` interface.

We can easily combine multiple fixtures. Here, as example, we make action with a counter that counts "visits".

```
from py4web import action, Session, Translator, DAL
from py4web.utils.dbstore import DBStore
import os
db = DAL('sqlite:memory')
session =  Session(storage=DBStore(db))
T_FOLDER = os.path.join(os.path.dirname(__file__), 'translations')
T = Translator(T_FOLDER)

@action('index')
@action.uses(session, T)
```

```
def index():
    counter = session.get('counter', -1)
    counter += 1
    session['counter'] = counter
    return str(T("You have been here {n} times").format(n=counter))
```

Now create the following translation file `translations/en.json`:

```
{"You have been here {n} times":
  {
    "0": "This your first time here",
    "1": "You have been here once before",
    "2": "You have been here twice before",
    "3": "You have been here {n} times",
    "6": "You have been here more than 5 times"
  }
}
```

When visiting this site with the browser language preference set to english and reloading multiple times you will get the following messages:

```
This your first time here
You have been here once before
You have been here twice before
You have been here 3 times
You have been here 4 times
You have been here 5 times
You have been here more than 5 times
```

Now try create a file called `translations/it.json` which contains:

```
{"You have been here {n} times":
  {
    "0": "Non ti ho mai visto prima",
    "1": "Ti ho gia' visto",
    "2": "Ti ho gia' visto 2 volte",
    "3": "Ti ho visto {n} volte",
    "6": "Ti ho visto piu' di 5 volte"
  }
}
```

and set your browser preference to Italian.

## 5.5 The Flash fixture

It is common to want to display "alerts" to the suers. Here we refer to them as flash messeges. There is a little more to it than just displaying a message to the view because flash messages can have state that must be preserved after redirection. Also they can be generated both server side and client side, there can be only one at the time, they may have a type, and they should be dismissible.

The Flash helper handles the server side of them. Here is an example:

```
from py4web import Flash

flash = Flash()
```

```
@action('index')
@action.uses(Flash)
def index():
    flash.set("Hello World", _class="info", sanitize=True)
    return dict()
```

and in the template:

```
...
<div id="py4web-flash"></div>
...
<script src="js/utils.js"></script>
[[if globals().get('flash'):]]<script>utils.flash([[=XML(flash)]]);</script>[[pass]]
```

By setting the value of the message in the flash helper, a flash variable is returned by the action and this trigger the JS in the template to inject the message in the `#py4web-flash` DIV which you can position at your convenience. Also the optional class is applied to the injected HTML.

If a page is redirected after a flash is set, the flash is remembered. This is achieved by asking the browser to keep the message temporarily in a one-time cookie. After redirection the message is sent back by the browser to the server and the server sets it again automatically before returning content, unless it is overwritten by another set.

The client can also set/add flash messages by calling:

```
utils.flash({'message': 'hello world', 'class': 'info'});
```

py4web defaults to an alert class called `default` and most CSS frameworks define classes for alerts called `success`, `error`, `warning`, `default`, and `info`. Yet, there is nothing in py4web that hardcodes those names. You can use your own class names.

## 5.6 The DAL fixture

We have already used the `DAL` fixture in the context of sessions but maybe you want direct access to the DAL object for the purpose of accessing the database, not just sessions.

PY4WEB, by default, uses the PyDAL (Python Database Abstraction Layer) which is documented in a later chapter. Here is an example, please remember to create the `databases` folder under your project in case it doesn't exist:

```
from datetime import datetime
from py4web import action, request, DAL, Field
import os

DB_FOLDER = os.path.join(os.path.dirname(__file__), 'databases')
db = DAL('sqlite://storage.db', folder=DB_FOLDER, pool_size=1)
db.define_table('visit_log', Field('client_ip'), Field('timestamp', 'datetime'))
db.commit()

@action('index')
@action.uses(db)
def index():
    client_ip = request.environ.get('REMOTE_ADDR')
    db.visit_log.insert(client_ip=client_ip, timestamp=datetime.utcnow())
    return "Your visit was stored in database"
```

Notice that the database fixture defines (creates/re-creates tables) automatically when py4web starts (and

every time it reloads this app) and picks a connection from the connection pool at every HTTP request. Also each call to the `index()` action is wrapped into a transaction and it commits `on_success` and rolls back `on_error`.

## 5.7 Caveats about Fixtures

Since fixtures are shared by multiple actions you are not allowed to change their state because it would not be thread safe. There is one exception to this rule. Actions can change some attributes of database fields:

```
from py4web import Field, action, request, DAL, Field
from py4web.utils.form import Form
import os

DB_FOLDER = os.path.join(os.path.dirname(__file__), 'databases')
db = DAL('sqlite://storage.db', folder=DB_FOLDER, pool_size=1)
db.define_table('thing', Field('name', writable=False))

@action('index')
@action.uses(db, 'generic.html')
def index():
    db.thing.name.writable = True
    form = Form(db.thing)
    return dict(form=form)
)
```

Note thas this code will only be able to display a form, to process it after submit, additional code needs to be added, as we will see later on. This example is assuming that you created the application from the scaffolding app, so that a generic.html is already created for you.

The `readable`, `writable`, `default`, `update`, and `require` attributes of `db.{table}.{field}` are special objects of class `ThreadSafeVariable` defined the `threadsafevariable` module. These objects are very much like Python thread local objects but they are re-initialized at every request using the value specified outside of the action. This means that actions can safely change the values of these attributes.

## 5.8 Custom fixtures

A fixture is an object with the following minimal structure:

```
from py4web import Fixture

class MyFixture(Fixture):
    def on_request(self): pass
    def on_success(self): pass
    def on_error(self): pass
    def transform(self, data): return data
```

if an action uses this fixture:

```
@action('index')
@action.uses(MyFixture())
def index(): return 'hello world'
```

Then `on_request()` is guaranteed to be called before the `index()` function is called. The `on_success()` is guaranteed to be called if the `index()` function returns successfully or raises `HTTP` or performs a `redirect`. The `on_error()` is guaranteed to be called when the `index()` function raises any exception other than `HTTP`. The `transform` function is called to perform any desired transformation of the value returned by the `index()` function.

## 5.9 Auth and Auth.user

`auth` and `auth.user` are both fixtures. They depend on `session`. The role of access is to provide the action with authentication information. It is used as follows:

```
from py4web import action, redirect, Session, DAL, URL
from py4web.utils.auth import Auth
import os

session = Session(secret='my secret key')
DB_FOLDER = os.path.join(os.path.dirname(__file__), 'databases')
db = DAL('sqlite://storage.db', folder=DB_FOLDER, pool_size=1)
auth = Auth(session, db)
auth.enable()

@action('index')
@action.uses(auth)
def index():
    user = auth.get_user() or redirect(URL('auth/login'))
    return 'Welcome %s' % user.get('first_name')
```

The constructor of the `Auth` object defines the `auth_user` table with the following fields: username, email, password, first_name, last_name, sso_id, and action_token (the last two are mostly for internal use).

`auth.enable()` registers multiple actions including `{appname}/auth/login` and it requires the presence of the `auth.html` template and the `auth` value component provided by the `_scaffold` app.

The `auth` object is the fixture. It manages the user information. It exposes a single method:

```
auth.get_user()
```

which returns a python dictionary containing the information of the currently logged in user. If the user is not logged-in, it returns `None`. The code of the example redirects to the 'auth/login' page if there is no user.

Since this check is very common, py4web provides an additional fixture `auth.user`:

```
@action('index')
@action.uses(auth.user)
def index():
    user = auth.get_user()
    return 'Welcome %s' % user.get('first_name')
```

This fixture automatically redirects to the `auth/login` page if user is not logged-in. It depends on `auth`, which depends on `db` and `session`.

The `Auth` fixture is plugin based and supports multiple plugin methods. They include Oauth2 (Google, Facebook, Twitter), PAM, LDAP, and SMAL2.

Here is an example of using the Google Oauth2 plugin:

```
from py4web.utils.auth_plugins.oauth2google import OAuth2Google
auth.register_plugin(OAuth2Google(
    client_id='...',
    client_secret='...',
    callback_url='auth/plugin/oauth2google/callback'))
```

The `client_id` and `client_secret` are provided by google. The callback url is the default option for py4web and it must be whitelisted with Google. All `Auth` plugins are objects. Different plugins are configured in different ways but they are registered using `auth.register_plugin(...)`. Examples are provided in `_scaffold/common.py`.

## 5.10 Caching and Memoize

py4web provides a cache in ram object that implements the Last Recently Used (LRU) Algorithm. It can be used to cache any function via a decorator:

```
import uuid
from py4web import Cache, action
cache = Cache(size=1000)

@action('hello/<name>')
@cache.memoize(expiration=60)
def hello(name):
    return "Hello %s your code is %s" % (name, uuid.uuid4())
```

It will cache (memoize) the return value of the `hello` function, as function of the input `name`, for up to 60 seconds. It will store in cache the 1000 most recently used values. The data is always stored in ram.

The Cache object is not a fixture and it should not and cannot be registered using the `@action.uses` object but we mention it here because some of the fixtures use this object internally. For example, template files are cached in ram to avoid accessing the file system every time a template needs to be rendered.

## 5.11 Convenience Decorators

The `_scaffold` application, in `common.py` defines two special convenienncе decorators:

```
@unauthenticated
def index():
    return dict()
```

and

```
`@authenticated def index():     return dict()
```

They apply all of the decorators below, use a template with the same name as the function (.html), and also register a route with the name of action followed the number of arguments of the action separated by a slash (/).

@unauthenticated does not require the user to be logged in. @authenticated required the user to be logged in.

If can be combined with (and precede) other `@action.uses(...)` but they should not be combined with `@action(...)` because they perform that function automatically.

# The database abstraction layer (DAL)

## 6.1 Dependencies

py4web comes with a Database Abstraction Layer (DAL), an API that maps Python objects into database objects such as queries, tables, and records. The DAL dynamically generates the SQL in real time using the specified dialect for the database back end, so that you do not have to write SQL code or learn different SQL dialects (the term SQL is used generically), and the application will be portable among different types of databases. A partial list of supported databases is show in the table below. Please check on the py4web web site and mailing list for more recent adapters. Google NoSQL is treated as a particular case in Chapter 13.

The Gotchas section at the end of this chapter has some more information about specific databases.

The Windows binary distribution works out of the box with SQLite, MSSQL, PostgreSQL and MySQL. The Mac binary distribution works out of the box with SQLite. To use any other database back-end, run from the source distribution and install the appropriate driver for the required back end.

Once the proper driver is installed, start py4web from source, and it will find the driver. Here is a list of the drivers py4web can use:

| database | drivers (source) |
|---|---|
| SQLite | sqlite3 or pysqlite2 or zxJDBC (on Jython) |
| PostgreSQL | psycopg2 or zxJDBC (on Jython) |
| MySQL | pymysql or MySQLdb |
| Oracle | cx_Oracle |
| MSSQL | pyodbc or pypyodbc |
| FireBird | kinterbasdb or fdb or pyodbc |
| DB2 | pyodbc |
| Informix | informixdb |
| Ingres | ingresdbi |
| Cubrid | cubriddb |
| Sybase | Sybase |
| Teradata | pyodbc |
| SAPDB | sapdb |
| MongoDB | pymongo |
| IMAP | imaplib |

`sqlite3`, `pymysql`, and `imaplib` ship with py4web. Support of MongoDB is experimental. The IMAP option allows to use DAL to access IMAP.

## 6.2 The DAL: A quick tour

py4web defines the following classes that make up the DAL:

The **DAL** object represents a database connection. For example:

```
db = DAL('sqlite://storage.sqlite')
```

**Table** represents a database table. You do not directly instantiate Table; instead, `DAL.define_table` instantiates it.

```
db.define_table('mytable', Field('myfield'))
```

The most important methods of a Table are:

`insert`, `truncate`, `drop`, and `import_from_csv_file`.

**Field** represents a database field. It can be instantiated and passed as an argument to `DAL.define_table`.

**DAL Rows** is the object returned by a database select. It can be thought of as a list of `Row` rows:

```
rows = db(db.mytable.myfield != None).select()
```

**Row** contains field values.

```
for row in rows:
    print row.myfield
```

**Query** is an object that represents a SQL "where" clause:

```
myquery = (db.mytable.myfield != None) | (db.mytable.myfield > 'A')
```

**Set** is an object that represents a set of records. Its most important methods are `count`, `select`, `update`, and `delete`. For example:

```
myset = db(myquery)
rows = myset.select()
myset.update(myfield='somevalue')
myset.delete()
```

**Expression** is something like an `orderby` or `groupby` expression. The Field class is derived from the Expression. Here is an example.

```
myorder = db.mytable.myfield.upper() | db.mytable.id
db().select(db.table.ALL, orderby=myorder)
```

## 6.3 Using the DAL "stand-alone"

The DAL can be used in a non-py4web environment via

```
from pydal import DAL, Field
```

## 6.4 DAL constructor

Basic use:

```
>>> db = DAL('sqlite://storage.sqlite')
```

The database is now connected and the connection is stored in the global variable `db`.

At any time you can retrieve the connection string.

```
>>> db._uri
sqlite://storage.sqlite
```

and the database name

```
>>> db._dbname
sqlite
```

The connection string is called a `_uri` because it is an instance of a Uniform Resource Identifier.

The DAL allows multiple connections with the same database or with different databases, even databases of different types. For now, we will assume the presence of a single database since this is the most common situation.

### 6.4.1 DAL signature

```
DAL(uri='sqlite://dummy.db',
    pool_size=0,
    folder=None,
    db_codec='UTF-8',
    check_reserved=None,
    migrate=True,
    fake_migrate=False,
    migrate_enabled=True,
    fake_migrate_all=False,
    decode_credentials=False,
    driver_args=None,
    adapter_args=None,
    attempts=5,
    auto_import=False,
    bigint_id=False,
    debug=False,
    lazy_tables=False,
    db_uid=None,
    do_connect=True,
    after_connection=None,
    tables=None,
    ignore_field_case=True,
    entity_quoting=False,
    table_hash=None)
```

### 6.4.2 Connection strings (the uri parameter)

A connection with the database is established by creating an instance of the DAL object:

```
db = DAL('sqlite://storage.sqlite', pool_size=0)
```

`db` is not a keyword; it is a local variable that stores the connection object `DAL`. You are free to give it a different name. The constructor of `DAL` requires a single argument, the connection string. The connection string is the only py4web code that depends on a specific back-end database. Here are examples of connection strings for specific types of supported back-end databases (in all cases, we assume the database is running from localhost on its default port and is named "test"):

| SQLite | **sqlite://storage.sqlite** |
|---|---|
| **MySQL** | `mysql://username:password@localhost/test?set_encoding=utf8mb4` |
| **PostgreSQL** | `postgres://username:password@localhost/test` |
| **MSSQL (legacy)** | `mssql://username:password@localhost/test` |
| **MSSQL (>=2005)** | `` `*mssql3://username:password@localhost/test*` `` |
| **MSSQL (>=2012)** | `` `*mssql4://username:password@localhost/test*` `` |
| **FireBird** | `firebird://username:password@localhost/test` |
| **Oracle** | `oracle://username/password@test` |
| **DB2** | `db2://username:password@test` |
| **Ingres** | `` `*ingres://username:password@localhost/test*` `` |
| **Sybase** | `` `*sybase://username:password@localhost/test*` `` |
| **Informix** | `informix://username:password@test` |
| **Teradata** | `teradata://DSN=dsn;UID=user;PWD=pass;DATABASE=test` |
| **Cubrid** | `` `*cubrid://username:password@localhost/test*` `` |
| **SAPDB** | `sapdb://username:password@localhost/test` |
| **IMAP** | `imap://user:password@server:port` |
| **MongoDB** | `` ``mongodb://username:password@localhost/test`` `` |
| **Google/SQL** | `google:sql://project:instance/database` |
| **Google/NoSQL** | `google:datastore` |
| **Google/NoSQL/NDB** | `google:datastore+ndb` |

Notice that in SQLite the database consists of a single file. If it does not exist, it is created. This file is locked every time it is accessed. In the case of MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2, Ingres and Informix the database "test" must be created outside py4web. Once the connection is established, py4web will create, alter, and drop tables appropriately.

In the MySQL connection string, the `?set_encoding=utf8mb4` at the end sets the encoding to UTF-8 and avoids an `Invalid utf8 character string:` error on Unicode characters that consist of four bytes, as by default, MySQL can only handle Unicode characters that consist of one to three bytes.

In the Google/NoSQL case the `+ndb` option turns on NDB. NDB uses a Memcache buffer to read data that is accessed often. This is completely automatic and done at the datastore level, not at the py4web level.

It is also possible to set the connection string to `None`. In this case DAL will not connect to any back-end database, but the API can still be accessed for testing.

Some times you may need to generate SQL as if you had a connection but without actually connecting to the database. This can be done with

```
db = DAL('...', do_connect=False)
```

In this case you will be able to call `_select`, `_insert`, `_update`, and `_delete` to generate SQL but not call `select`, `insert`, `update`, and `delete`. In most of the cases you can use `do_connect=False` even without having the required database drivers.

Notice that by default py4web uses utf8 character encoding for databases. If you work with existing databases that behave differently, you have to change it with the optional parameter `db_codec` like

```
db = DAL('...', db_codec='latin1')
```

Otherwise you'll get UnicodeDecodeError tickets.

### 6.4.3 Connection pooling

A common argument of the DAL constructor is the `pool_size`; it defaults to zero.

As it is rather slow to establish a new database connection for each request, py4web implements a mechanism for connection pooling. Once a connection is established and the page has been served and the transaction completed, the connection is not closed but goes into a pool. When the next http request arrives, py4web tries to recycle a connection from the pool and use that for the new transaction. If there are no available connections in the pool, a new connection is established.

When py4web starts, the pool is always empty. The pool grows up to the minimum between the value of `pool_size` and the max number of concurrent requests. This means that if `pool_size=10` but our server never receives more than 5 concurrent requests, then the actual pool size will only grow to 5. If `pool_size=0` then connection pooling is not used.

Connections in the pools are shared sequentially among threads, in the sense that they may be used by two different but not simultaneous threads. There is only one pool for each py4web process.

The `pool_size` parameter is ignored by SQLite and Google App Engine. Connection pooling is ignored for SQLite, since it would not yield any benefit.

### 6.4.4 Connection failures (attempts parameter)

If py4web fails to connect to the database it waits 1 second and by default tries again up to 5 times before declaring a failure. In case of connection pooling it is possible that a pooled connection that stays open but unused for some time is closed by the database end. Thanks to the retry feature py4web tries to re-establish these dropped connections. The number of attempts is set via the attempts parameter.

### 6.4.5 Lazy Tables

setting `lazy_tables = True` provides a major performance boost. See below: lazy tables

### 6.4.6 Model-less applications

Using py4web's model directory for your application models is very convenient and productive. With lazy tables and conditional models, performance is usually acceptable even for large applications. Many experienced developers use this in production environments.

However, it is possible to define DAL tables on demand inside controller functions or modules. This may make sense when the number or complexity of table definitions overloads the use of lazy tables and conditional models.

This is referred to as "model-less" development by the py4web community. It means less use of the automatic execution of Python files in the model directory. It does not imply abandoning the concept of models, views and controllers.

PY4WEB's auto-execution of Python code inside the model directory does this for you:

- models are run automatically every time a request is processed
- models access py4web's global scope.

Models also make for useful interactive shell sessions when py4web is started with the -M commandline option.

Also, remember maintainability: other py4web developers expect to find model definitions in the model directory.

To use the "model-less" approach, you take responsibility for doing these two housekeeping tasks. You call the table definitions when you need them, and provide necessary access passed as parameter.

For example, a typical model-less application may leave the definitions of the database connection objects in the model file, but define the tables on demand per controller function.

The typical case is to move the table definitions to a module file (a Python file saved in the modules directory).

If the function to define a set of tables is called `define_employee_tables()` in a module called "table_setup.py", your controller that wants to refer to the tables related to employee records in order to make an SQLFORM needs to call the `define_employee_tables()` function before accessing any tables. The `define_employee_tables()` function needs to access the database connection object in order to define tables. You need to pass the db object to the `define_employee_tables()` (as mentioned above).

### 6.4.7 Replicated databases

The first argument of `DAL(...)` can be a list of URIs. In this case py4web tries to connect to each of them. The main purpose for this is to deal with multiple database servers and distribute the workload among them). Here is a typical use case:

```
db = DAL(['mysql://...1', 'mysql://...2', 'mysql://...3'])
```

In this case the DAL tries to connect to the first and, on failure, it will try the second and the third. This can also be used to distribute load in a database master-slave configuration.

### 6.4.8 Reserved keywords

`check_reserved` tells the constructor to check table names and column names against reserved SQL keywords in target back-end databases. `check_reserved` defaults to None.

This is a list of strings that contain the database back-end adapter names.

The adapter name is the same as used in the DAL connection string. So if you want to check against PostgreSQL and MSSQL then your connection string would look as follows:

```
db = DAL('sqlite://storage.sqlite', check_reserved=['postgres', 'mssql'])
```

The DAL will scan the keywords in the same order as of the list.

There are two extra options "all" and "common". If you specify all, it will check against all known SQL keywords. If you specify common, it will only check against common SQL keywords such as `SELECT`, `INSERT`, `UPDATE`, etc.

For supported back-ends you may also specify if you would like to check against the non-reserved SQL keywords as well. In this case you would append `_nonreserved` to the name. For example:

```
check_reserved=['postgres', 'postgres_nonreserved']
```

The following database backends support reserved words checking.

| PostgreSQL | `postgres(_nonreserved)` |
|---|---|
| MySQL | `mysql` |
| FireBird | `firebird(_nonreserved)` |
| MSSQL | `mssql` |
| Oracle | `oracle` |

### 6.4.9 Database quoting and case settings

Quoting of SQL entities are enabled by default in DAL, that is:

```
entity_quoting = True
```

This way identifiers are automatically quoted in SQL generated by DAL. At SQL level keywords and unquoted identifiers are case insensitive, thus quoting an SQL identifier makes it case sensitive.

> Notice that unquoted identifiers should always be folded to lower case by the back-end engine according to SQL standard but not all engines are compliant with this (for example PostgreSQL default folding is upper case).

By default DAL ignores field case too, to change this use:

```
ignore_field_case = False
```

To be sure of using the same names in python and in the DB schema, you must arrange for both settings above. Here is an example:

```
db = DAL(ignore_field_case=False)
db.define_table('table1', Field('column'), Field('COLUMN'))
query = db.table1.COLUMN != db.table1.column
```

### 6.4.10 Making a secure connection

Sometimes it is necessary (and advised) to connect to your database using secure connection, especially if your database is not on the same server as your application. In this case you need to pass additional parameters to the database driver. You should refer to database driver documentation for details.

For PostgreSQL with psycopg2 it should look like this:

```
DAL('postgres://user_name:user_password@server_addr/db_name',
    driver_args={'sslmode': 'require', 'sslrootcert': 'root.crt',
                 'sslcert': 'postgresql.crt', 'sslkey': 'postgresql.key'})
```

where parameters `sslrootcert`, `sslcert` and `sslkey` should contain the full path to the files. You should refer to PostgreSQL documentation on how to configure PostgreSQL server to accept secure connections.

### 6.4.11 Other DAL constructor parameters

#### Database folder location

`folder` sets the place where migration files will be created (see Migrations section in this chapter for details). It is also used for SQLite databases. Automatically set within py4web. Set a path when using DAL outside py4web.

#### Default migration settings

The DAL constructor migration settings are booleans affecting defaults and global behaviour.

`migrate = True` sets default migrate behavior for all tables

`fake_migrate = False` sets default fake_migrate behavior for all tables

`migrate_enabled = True` If set to False disables ALL migrations

`fake_migrate_all = False` If set to True fake migrates ALL tables

### 6.4.12 Experiment with the py4web shell

You can experiment with the DAL API using the py4web shell, that is available using the `shell` command (read more in Chapter 1).

> You need to choose an application to run the shell on, mind that database changes may be persistent. So be carefull and do NOT exitate to create a new application for doing testing instead of tampering with an existing one.

Start by creating a connection. For the sake of example, you can use SQLite. Nothing in this discussion changes when you change the back-end engine.

Note that most of the code snippets that contain the python prompt `>>>` are directly executable via a plain shell, which you can obtain using `-PS` command line options.

## 6.5 Table constructor

Tables are defined in the DAL via `define_table`.

### 6.5.1 define_table signature

The signature for define_table method is:

```
define_table(tablename, *fields, **kwargs)
```

It accepts a mandatory table name and an optional number of `Field` instances (even none). You can also pass a `Table` (or subclass) object instead of a `Field` one, this clones and adds all the fields (but the "id") to the defining table. Other optional keyword args are: `rname`, `redefine`, `common_filter`, `fake_migrate`, `fields`, `format`, `migrate`, `on_define`, `plural`, `polymodel`, `primarykey`, `sequence_name`, `singular`, `table_class`, and `trigger_name`, which are discussed below.

For example:

```
>>> db.define_table('person', Field('name'))
<Table person (id, name)>
```

It defines, stores and returns a `Table` object called "person" containing a field (column) "name". This object can also be accessed via `db.person`, so you do not need to catch the value returned by define_table.

### 6.5.2 `id`: Notes about the primary key

Do not declare a field called "id", because one is created by py4web anyway. Every table has a field called "id" by default. It is an auto-increment integer field (usually starting at 1) used for cross-reference and for making every record unique, so "id" is a primary key. (Note: the id counter starting at 1 is back-end specific. For example, this does not apply to the Google App Engine NoSQL.)

Optionally you can define a field of `type='id'` and py4web will use this field as auto-increment id field. This is not recommended except when accessing legacy database tables which have a primary key under a different name. With some limitation, you can also use different primary keys using the `primarykey` parameter.

### 6.5.3 `plural` and `singular`

As pydal is a general DAL, it includes plural and singular attributes to refer to the table names so that external elements can use the proper name for a table. A use case is in web2py with Smartgrid objects with references to external tables.

### 6.5.4 `redefine`

Tables can be defined only once but you can force py4web to redefine an existing table:

```
db.define_table('person', Field('name'))
db.define_table('person', Field('name'), redefine=True)
```

The redefinition may trigger a migration if table definition changes.

### 6.5.5 `format`: Record representation

It is optional but recommended to specify a format representation for records with the `format` parameter.

```
db.define_table('person', Field('name'), format='%(name)s')
```

or

```
db.define_table('person', Field('name'), format='%(name)s %(id)s')
```

or even more complex ones using a function:

```
db.define_table('person', Field('name'),
                format=lambda r: r.name or 'anonymous')
```

The format attribute will be used for two purposes: - To represent referenced records in select/option drop-downs. - To set the `db.othertable.otherfield.represent` attribute for all fields referencing this table. This means that the `Form` constructor will not show references by id but will use the preferred format representation instead.

### 6.5.6 `rname`: Real name

`rname` sets a database backend name for the table. This makes the py4web table name an alias, and `rname` is the real name used when constructing the query for the backend. To illustrate just one use, `rname` can be used to provide MSSQL fully qualified table names accessing tables belonging to other databases on the server: `rname = 'db1.dbo.table1'`:python

### 6.5.7 `primarykey`: Support for legacy tables

`primarykey` helps support legacy tables with existing primary keys, even multi-part. See Legacy databases and keyed tables section in this chapter.

### 6.5.8 `migrate`, `fake_migrate`

`migrate` sets migration options for the table. Refer to Migrations section in this chapter for details.

### 6.5.9 table_class

If you define your own Table class as a sub-class of pydal.objects.Table, you can provide it here; this allows you to extend and override methods. Example:

```
from pydal.objects import Table

class MyTable(Table):
    ...

db.define_table(..., table_class=MyTable)
```

### 6.5.10 sequence_name

The name of a custom table sequence (if supported by the database). Can create a SEQUENCE (starting at 1 and incrementing by 1) or use this for legacy tables with custom sequences.

Note that when necessary, py4web will create sequences automatically by default.

### 6.5.11 trigger_name

Relates to `sequence_name`. Relevant for some backends which do not support auto-increment numeric fields.

### 6.5.12 polymodel

For Google App Engine

### 6.5.13 on_define

`on_define` is a callback triggered when a lazy_table is instantiated, although it is called anyway if the table is not lazy. This allows dynamic changes to the table without losing the advantages of delayed instantiation.

Example:

```
db = DAL(lazy_tables=True)
db.define_table('person',
    Field('name'),
    Field('age', 'integer'),
    on_define=lambda table: [
        table.name.set_attributes(requires=IS_NOT_EMPTY(), default=''),
        table.age.set_attributes(requires=IS_INT_IN_RANGE(0, 120), default=30) ])
```

Note this example shows how to use `on_define` but it is not actually necessary. The simple `requires` values could be added to the Field definitions and the table would still be lazy. However, `requires` which take a Set object as the first argument, such as IS_IN_DB, will make a query like `db.sometable.somefield == some_value` which would cause `sometable` to be defined early. This is the situation saved by `on_define`.

### 6.5.14 Lazy Tables, a major performance boost

py4web models are executed before controllers, so all tables are defined at every request. Not all tables are needed to handle each request, so it is possible that some of the time spent defining tables is wasted. Conditional models (see Model-less applications) can help, but py4web offers a big performance boost via lazy_tables. This feature means that table creation is deferred until the table is actually referenced.

Enabling lazy tables is made when initialising a database via the DAL constructor. It requires setting the lazy_tables parameter: `DAL(..., lazy_tables=True)`:python This is one of the most significant response-time performance boosts in py4web.

### 6.5.15 Adding attributes to fields and tables

If you need to add custom attributes to fields, you can simply do this: `db.table.field.extra = {}`

"extra" is not a keyword ; it's a custom attributes now attached to the field object. You can do it with tables too but they must be preceded by an underscore to avoid naming conflicts with fields:

`db.table._extra = {}`:python

## 6.6 Field constructor

These are the default values of a Field constructor:

```
Field(fieldname, type='string', length=None, default=DEFAULT,
      required=False, requires=DEFAULT,
      ondelete='CASCADE', notnull=False, unique=False,
      uploadfield=True, widget=None, label=None, comment=None,
      writable=True, readable=True, searchable=True, listable=True,
      update=None, authorize=None, autodelete=False, represent=None,
      uploadfolder=None, uploadseparate=None, uploadfs=None,
      compute=None, filter_in=None, filter_out=None,
      custom_qualifier=None, map_none=None, rname=None)
```

where DEFAULT is a special value used to allow the value None for a parameter.

Not all of them are relevant for every field. `length` is relevant only for fields of type "string". `uploadfield`, `authorize`, and `autodelete` are relevant only for fields of type "upload". `ondelete` is relevant only for fields of type "reference" and "upload".

- `length` sets the maximum length of a "string", "password" or "upload" field. If `length` is not specified a default value is used but the default value is not guaranteed to be backward compatible. *To avoid unwanted migrations on upgrades, we recommend that you always specify the length for string, password and upload fields.*
- `default` sets the default value for the field. The default value is used when performing an insert if a value is not explicitly specified. It is also used to pre-populate forms built from the table using `Form`. Note, rather than being a fixed value, the default can instead be a function (including a lambda function) that returns a value of the appropriate type for the field. In that case, the function is called once for each record inserted, even when multiple records are inserted in a single transaction.
- `required` tells the DAL that no insert should be allowed on this table if a value for this field is not explicitly specified.
- `requires` is a validator or a list of validators. This is not used by the DAL, but it is used by `Form`. The default validators for the given types are shown in the next section.

  Notice that while `requires=...` is enforced at the level of forms, `required=True` is enforced at the level of the DAL (insert). In addition, `notnull`, `unique` and `ondelete` are enforced at the level of the database. While they sometimes may seem redundant, it is important to maintain the distinction when programming with the DAL.

- `rname` provides the field with a "real name", a name for the field known to the database adapter; when the field is used, it is the rname value which is sent to the database. The py4web name for the field is then effectively an alias.

- `ondelete` translates into the "ON DELETE" SQL statement. By default it is set to "CASCADE". This tells the database that when it deletes a record, it should also delete all records that refer to it. To disable this feature, set `ondelete` to "NO ACTION" or "SET NULL".
- `notnull=True` translates into the "NOT NULL" SQL statement. It prevents the database from inserting null values for the field.
- `unique=True` translates into the "UNIQUE" SQL statement and it makes sure that values of this field are unique within the table. It is enforced at the database level.
- `uploadfield` applies only to fields of type "upload". A field of type "upload" stores the name of a file saved somewhere else, by default on the filesystem under the application "uploads/" folder. If `uploadfield` is set to True, then the file is stored in a blob field within the same table and the value of `uploadfield` is the name of the blob field. This will be discussed in more detail later in the *More on uploads* section in this chapter.
- `uploadfolder` sets the folder for uploaded files. By default, an uploaded file goes into the application's "uploads/" folder, that is into `os.path.join(request.folder, 'uploads')` (this seems not the case for MongoAdapter at present). For example: `Field(..., uploadfolder=os.path.join(request.folder, 'static/temp'))`:python will upload files to the "py4web/applications/myapp/static/temp" folder.
- `uploadseparate` if set to True will upload files under different subfolders of the *uploadfolder* folder. This is optimized to avoid too many files under the same folder/subfolder. ATTENTION: You cannot change the value of `uploadseparate` from True to False without breaking links to existing uploads. py4web either uses the separate subfolders or it does not. Changing the behavior after files have been uploaded will prevent py4web from being able to retrieve those files. If this happens it is possible to move files and fix the problem but this is not described here.
- `uploadfs` allows you specify a different file system where to upload files, including an Amazon S3 storage or a remote SFTP storage.

  You need to have PyFileSystem installed for this to work. `uploadfs` must point to PyFileSystem.
- `autodelete` determines if the corresponding uploaded file should be deleted when the record referencing the file is deleted. For "upload" fields only. However, records deleted by the database itself due to a CASCADE operation will not trigger py4web's autodelete. The py4web Google group has workaround discussions.
- `widget` must be one of the available widget objects, including custom widgets, for example: `SQLFORM.widgets.string.widget`. A list of available widgets will be discussed later. Each field type has a default widget.
- `label` is a string (or a helper or something that can be serialized to a string) that contains the label to be used for this field in auto-generated forms.
- `comment` is a string (or a helper or something that can be serialized to a string) that contains a comment associated with this field, and will be displayed to the right of the input field in the auto-generated forms.
- `writable` declares whether a field is writable in forms.
- `readable` declares whether a field is readable in forms. If a field is neither readable nor writable, it will not be displayed in create and update forms.
- `searchable` declares whether a field is searchable in grids (`SQLFORM.grid` and `SQLFORM.smartgrid` are described in *Chapter 7 ../07* ). Notice that a field must also be readable to be searched.
- `listable` declares whether a field is visible in grids (when listing multiple records)
- `update` contains the default value for this field when the record is updated.

- `compute` is an optional function. If a record is inserted or updated, the compute function will be executed and the field will be populated with the function result. The record is passed to the compute function as a `dict`, and the dict will not include the current value of that, or any other compute field.
- `authorize` can be used to require access control on the corresponding field, for "upload" fields only. It will be discussed more in detail in the context of Authentication and Authorization.
- `represent` can be None or can point to a function that takes a field value and returns an alternate representation for the field value. Examples:

```
db.mytable.name.represent = lambda name, row: name.capitalize()
db.mytable.other_id.represent = lambda oid, row: row.myfield
db.mytable.some_uploadfield.represent = lambda val, row: A('get it',
_href=URL('download', args=val))
```

- `filter_in` and `filter_out` can be set to callables for further processing of field's value. `filter_in` is passed the field's value to be written to the database before an insert or update while `filter_out` is passed the value retrieved from the database before field assignment. The value returned by the callable is then used. See filter_in and filter_out section in this chapter.
- `custom_qualifier` is a custom SQL qualifier for the field to be used at table creation time (cannot use for field of type "id", "reference", or "big-reference").

### 6.6.1 Field types

| field type | default field validators |
|---|---|
| `string` | `IS_LENGTH(length)` default length is 512 |
| `text` | `IS_LENGTH(length)` default length is 32768 |
| `blob` | `None` default length is 2**31 (2 GiB) |
| `boolean` | `None` |
| `integer` | `IS_INT_IN_RANGE(-2**31, 2**31)` |
| `double` | `IS_FLOAT_IN_RANGE(-1e100, 1e100)` |
| `decimal(n,m)` | `` IS_DECIMAL_IN_RANGE(-10**10, 10**10)`` |
| `date` | `IS_DATE()` |
| `time` | `IS_TIME()` |
| `datetime` | `IS_DATETIME()` |
| `password` | `IS_LENGTH(length)` default length is 512 |
| `upload` | `None` default length is 512 |
| `reference <table>` | `IS_IN_DB(db, table.field, format)` |
| `list:string` | `None` |
| `list:integer` | `None` |
| `list:reference <table>` | `IS_IN_DB(d b, table._id, format, multiple=True)` |
| `json` | `IS_EMPTY_OR(IS_JSON())` default length is 512 |
| `bigint` | `IS_INT_IN_RANGE(-2**63, 2**63)` |
| `big-id` | `None` |
| `big-reference` | `None` |

Decimal requires and returns values as `Decimal` objects, as defined in the Python `decimal` module. SQLite does not handle the `decimal` type so internally we treat it as a `double`. The (n,m) are the number of digits in total and the number of digits after the decimal point respectively.

The `big-id` and, `big-reference` are only supported by some of the database engines and are experimental. They are not normally used as field types unless for legacy tables, however, the DAL constructor has a `bigint_id` argument that when set to `True` makes the `id` fields and `reference` fields `big-id` and `big-reference` respectively.

The `list:<type>` fields are special because they are designed to take advantage of certain denormalization features on NoSQL (in the case of Google App Engine NoSQL, the field types `ListProperty` and `StringListProperty`) and back-port them all the other supported relational databases. On relational databases lists are stored as a `text` field. The items are separated by a `|` and each `|` in string item is escaped as a `||`. They are discussed in *list: and contains* section in this chapter.

The `json` field type is pretty much explanatory. It can store any json serializable object. It is designed to work specifically for MongoDB and backported to the other database adapters for portability.

`blob` fields are also special. By default, binary data is encoded in base64 before being stored into the actual database field, and it is decoded when extracted. This has the negative effect of using 33% more storage space than necessary in blob fields, but has the advantageof making the communication independent of back-end-specific escaping conventions.

### 6.6.2 Run-time field and table modification

Most attributes of fields and tables can be modified after they are defined:

```
>>> db.define_table('person', Field('name', default=''), format='%(name)s')
<Table person (id, name)>
>>> db.person._format = '%(name)s/%(id)s'
>>> db.person.name.default = 'anonymous'
```

notice that attributes of tables are usually prefixed by an underscore to avoid conflict with possible field names.

You can list the tables that have been defined for a given database connection:

```
>>> db.tables
['person']
```

You can query for the type of a table:

```
>>> type(db.person)
<class 'pydal.objects.Table'>
```

You can access a table using different syntaxes:

```
>>> db.person is db['person']
True
```

You can also list the fields that have been defined for a given table:

```
>>> db.person.fields
['id', 'name']
```

Similarly you can access fields from their name in multiple equivalent ways:

```
>>> type(db.person.name)
<class 'pydal.objects.Field'>
>>> db.person.name is db.person['name']
True
```

Given a field, you can access the attributes set in its definition:

```
>>> db.person.name.type
string
>>> db.person.name.unique
False
>>> db.person.name.notnull
False
>>> db.person.name.length
32
```

including its parent table, tablename, and parent connection:

```
>>> db.person.name._table == db.person
True
>>> db.person.name._tablename == 'person'
True
>>> db.person.name._db == db
True
```

A field also has methods. Some of them are used to build queries and we will see them later. A special method of the field object is `validate` and it calls the validators for the field.

```
>>> db.person.name.validate('John')
('John', None)
```

which returns a tuple `(value, error)`. `error` is `None` if the input passes validation.

## 6.7 Migrations

`define_table` checks whether or not the corresponding table exists. If it does not, it generates the SQL to create it and executes the SQL. If the table does exist but differs from the one being defined, it generates the SQL to alter the table and executes it. If a field has changed type but not name, it will try to convert the data (If you do not want this, you need to redefine the table twice, the first time, letting py4web drop the field by removing it, and the second time adding the newly defined field so that py4web can create it.). If the table exists and matches the current definition, it will leave it alone. In all cases it will create the `db.person` object that represents the table.

We refer to this behavior as a "migration". py4web logs all migrations and migration attempts in the file "sql.log".

> Notice that by default py4web uses the "app/databases" folder for the log file and all other migration files it needs. You can change this setting the `folder` argument to DAL. To set a different log file name, for example "migrate.log" you can do `db = DAL(..., adapter_args=dict(logfile='migrate.log'))`:python

The first argument of `define_table` is always the table name. The other unnamed arguments are the fields (Field). The function also takes an optional keyword argument called "migrate":

```
db.define_table('person', ..., migrate='person.table')
```

The value of migrate is the filename where py4web stores internal migration information for this table. These files are very important and should never be removed while the corresponding tables exist. In cases where a table has been dropped and the corresponding file still exist, it can be removed manually. By default, migrate is set to True. This causes py4web to generate the filename from a hash of the connection string. If migrate is set to False, the migration is not performed, and py4web assumes that the table exists in the datastore and it contains (at least) the fields listed in `define_table`.

There may not be two tables in the same application with the same migrate filename.

The DAL class also takes a "migrate" argument, which determines the default value of migrate for calls to `define_table`. For example,

```
db = DAL('sqlite://storage.sqlite', migrate=False)
```

will set the default value of migrate to False whenever `db.define_table` is called without a migrate argument.

> Notice that py4web only migrates new columns, removed columns, and changes in column type (except in SQLite). py4web does not migrate changes in attributes such as changes in the values of `default`, `unique`, `notnull`, and `ondelete`.

Migrations can be disabled for all tables at once:

```
db = DAL(..., migrate_enabled=False)
```

This is the recommended behavior when two apps share the same database. Only one of the two apps should perform migrations, the other should disabled them.

### 6.7.1 Fixing broken migrations

There are two common problems with migrations and there are ways to recover from them.

One problem is specific with SQLite. SQLite does not enforce column types and cannot drop columns. This means that if you have a column of type string and you remove it, it is not really removed. If you add the column again with a different type (for example datetime) you end up with a datetime column that contains strings (junk for practical purposes). py4web does not complain about this because it does not know what is in the database, until it tries to retrieve records and fails.

If py4web returns an error in some parse function when selecting records, most likely this is due to corrupted data in a column because of the above issue.

The solution consists in updating all records of the table and updating the values in the column in question with None.

The other problem is more generic but typical with MySQL. MySQL does not allow more than one ALTER TABLE in a transaction. This means that py4web must break complex transactions into smaller ones (one ALTER TABLE at the time) and commit one piece at the time. It is therefore possible that part of a complex transaction gets committed and one part fails, leaving py4web in a corrupted state. Why would part of a transaction fail? Because, for example, it involves altering a table and converting a string column into a datetime column, py4web tries to convert the data, but the data cannot be converted. What happens to py4web? It gets confused about what exactly is the table structure actually stored in the database.

The solution consists of enabling fake migrations:

```
db.define_table(...., migrate=True, fake_migrate=True)
```

This will rebuild py4web metadata about the table according to the table definition. Try multiple table definitions to see which one works (the one before the failed migration and the one after the failed migration). Once successful remove the `fake_migrate=True` parameter.

Before attempting to fix migration problems it is prudent to make a copy of "applications/yourapp/databases/*.table" files.

Migration problems can also be fixed for all tables at once:

```
db = DAL(..., fake_migrate_all=True)
```

This also fails if the model describes tables that do not exist in the database, but it can help narrowing

down the problem.

### 6.7.2 Migration control summary

The logic of the various migration arguments are summarized in this pseudo-code:

```
if DAL.migrate_enabled and table.migrate:
   if DAL.fake_migrate_all or table.fake_migrate:
       perform fake migration
   else:
       perform migration
```

## 6.8 `insert`

Given a table, you can insert records

```
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
```

Insert returns the unique "id" value of each record inserted.

You can truncate the table, i.e., delete all records and reset the counter of the id.

```
>>> db.person.truncate()
```

Now, if you insert a record again, the counter starts again at 1 (this is back-end specific and does not apply to Google NoSQL):

```
>>> db.person.insert(name="Alex")
1
```

Notice you can pass a parameter to `truncate`, for example you can tell SQLite to restart the id counter.

```
>>> db.person.truncate('RESTART IDENTITY CASCADE')
```

The argument is in raw SQL and therefore engine specific.

py4web also provides a bulk_insert method

```
>>> db.person.bulk_insert([{'name': 'Alex'}, {'name': 'John'}, {'name': 'Tim'}])
[3, 4, 5]
```

It takes a list of dictionaries of fields to be inserted and performs multiple inserts at once. It returns the list of "id" values of the inserted records. On the supported relational databases there is no advantage in using this function as opposed to looping and performing individual inserts but on Google App Engine NoSQL, there is a major speed advantage.

## 6.9 `commit` and `rollback`

The insert, truncate, delete, and update operations aren't actually committed until py4web issues the commit command. The create and drop operations may be executed immediately, depending on the database engine. Calls to py4web actions are automatically wrapped in transactions. If you executed

commands via the shell, you are required to manually commit:

```
>>> db.commit()
```

To check it let's insert a new record:

```
>>> db.person.insert(name="Bob")
2
```

and roll back, i.e., ignore all operations since the last commit:

```
>>> db.rollback()
```

If you now insert again, the counter will again be set to 2, since the previous insert was rolled back.

```
>>> db.person.insert(name="Bob")
2
```

Code in models, views and controllers is enclosed in py4web code that looks like this (pseudo code) :

```
try:
    execute models, controller function and view
except:
    rollback all connections
    log the traceback
    send a ticket to the visitor
else:
    commit all connections
    save cookies, sessions and return the page
```

So in models, views and controllers there is no need to ever call `commit` or `rollback` explicitly in py4web unless you need more granular control. However, in modules you will need to use `commit()`.

## 6.10 Raw SQL

### 6.10.1 Timing queries

All queries are automatically timed by py4web. The variable `db._timings` is a list of tuples. Each tuple contains the raw SQL query as passed to the database driver and the time it took to execute in seconds. This variable can be displayed in views using the toolbar:

```
{{=response.toolbar()}}
```

### 6.10.2 `executesql`

The DAL allows you to explicitly issue SQL statements.

```
>>> db.executesql('SELECT * FROM person;')
[(1, u'Massimo'), (2, u'Massimo')]
```

In this case, the return values are not parsed or transformed by the DAL, and the format depends on the specific database driver. This usage with selects is normally not needed, but it is more common with indexes.

`executesql` takes five optional arguments: `placeholders`, `as_dict`, `fields`, `colnames`, and

as_ordered_dict.

placeholders is an optional sequence of values to be substituted in or, if supported by the DB driver, a dictionary with keys matching named placeholders in your SQL.

If as_dict is set to True, the results cursor returned by the DB driver will be converted to a sequence of dictionaries keyed with the db field names. Results returned with as_dict = True are the same as those returned when applying **.as_list()** to a normal select:

```
[{'field1': val1_row1, 'field2': val2_row1}, {'field1': val1_row2, 'field2':
val2_row2}]
```

as_ordered_dict is pretty much like as_dict but the former ensures that the order of resulting fields (OrderedDict keys) reflect the order on which they are returned from DB driver:

```
[OrderedDict([('field1', val1_row1), ('field2', val2_row1)]),
 OrderedDict([('field1', val1_row2), ('field2', val2_row2)])]
```

The fields argument is a list of DAL Field objects that match the fields returned from the DB. The Field objects should be part of one or more Table objects defined on the DAL object. The fields list can include one or more DAL Table objects in addition to or instead of including Field objects, or it can be just a single table (not in a list). In that case, the Field objects will be extracted from the table(s).

Instead of specifying the fields argument, the colnames argument can be specified as a list of field names in tablename.fieldname format. Again, these should represent tables and fields defined on the DAL object.

It is also possible to specify both fields and the associated colnames. In that case, fields can also include DAL Expression objects in addition to Field objects. For Field objects in "fields", the associated colnames must still be in tablename.fieldname format. For Expression objects in fields, the associated colnames can be any arbitrary labels.

Notice, the DAL Table objects referred to by fields or colnames can be dummy tables and do not have to represent any real tables in the database. Also, note that the fields and colnames must be in the same order as the fields in the results cursor returned from the DB.

### 6.10.3 _lastsql

Whether SQL was executed manually using executesql or was SQL generated by the DAL, you can always find the SQL code in db._lastsql. This is useful for debugging purposes:

```
>>> rows = db().select(db.person.ALL)
>>> db._lastsql
SELECT person.id, person.name FROM person;
```

> py4web never generates queries using the "*" operator. py4web is always explicit when selecting fields.

## 6.11 drop

Finally, you can drop tables and all data will be lost:

```
db.person.drop()
```

## 6.12 Indexes

Currently the DAL API does not provide a command to create indexes on tables, but this can be done using the `executesql` command. This is because the existence of indexes can make migrations complex, and it is better to deal with them explicitly. Indexes may be needed for those fields that are used in recurrent queries.

Here is an example of how to:

```
db = DAL('sqlite://storage.sqlite')
db.define_table('person', Field('name'))
db.executesql('CREATE INDEX IF NOT EXISTS myidx ON person (name);')
```

Other database dialects have very similar syntaxes but may not support the optional "IF NOT EXISTS" directive.

## 6.13 Legacy databases and keyed tables

py4web can connect to legacy databases under some conditions.

The easiest way is when these conditions are met: - Each table must have a unique auto-increment integer field called "id" - Records must be referenced exclusively using the "id" field.

When accessing an existing table, i.e., a table not created by py4web in the current application, always set `migrate=False`.

If the legacy table has an auto-increment integer field but it is not called "id", py4web can still access it but the table definition must declare the auto-increment field with 'id' type (that is using `FIeld('...', 'id')`).

Finally if the legacy table uses a primary key that is not an auto-increment id field it is possible to use a "keyed table", for example:

```
db.define_table('account',
                Field('accnum', 'integer'),
                Field('acctype'),
                Field('accdesc'),
                primarykey=['accnum', 'acctype'],
                migrate=False)
```

- `primarykey` is a list of the field names that make up the primary key.
- All primarykey fields have a `NOT NULL` set even if not specified.
- Keyed tables can only reference other keyed tables.
- Referencing fields must use the `reference tablename.fieldname` format.
- The `update_record` function is not available for Rows of keyed tables.

  Currently keyed tables are only supported for DB2, MSSQL, Ingres and Informix, but others engines will be added.

At the time of writing, we cannot guarantee that the `primarykey` attribute works with every existing legacy table and every supported database backend. For simplicity, we recommend, if possible, creating a database view that has an auto-increment id field.

## 6.14 Distributed transaction

> At the time of writing this feature is only supported by PostgreSQL, MySQL and Firebird, since they expose API for two-phase commits.

Assuming you have two (or more) connections to distinct PostgreSQL databases, for example:

```
db_a = DAL('postgres://...')
db_b = DAL('postgres://...')
```

In your models or controllers, you can commit them concurrently with:

```
DAL.distributed_transaction_commit(db_a, db_b)
```

On failure, this function rolls back and raises an `Exception`.

In controllers, when one action returns, if you have two distinct connections and you do not call the above function, py4web commits them separately. This means there is a possibility that one of the commits succeeds and one fails. The distributed transaction prevents this from happening.

## 6.15 More on uploads

Consider the following model:

```
db.define_table('myfile',
                Field('image', 'upload', default='path/to/file'))
```

In the case of an "upload" field, the default value can optionally be set to a path (an absolute path or a path relative to the current app folder), the default value is then assigned to each new record that does not specify an image.

Notice that this way multiple records may end to reference the same default image file and this could be a problem on a Field having `autodelete` enabled. When you do not want to allow duplicates for the image field (i.e. multiple records referencing the same file) but still want to set a default value for the "upload" then you need a way to copy the default file for each new record that does not specify an image. This can be obtained using a file-like object referencing the default file as the `default` argument to Field, or even with:

```
Field('image', 'upload', default=dict(data='<file_content>', filename='<file_name>'))
```

Normally an insert is handled automatically via a `Form` but occasionally you already have the file on the filesystem and want to upload it programmatically. This can be done in this way:

```
with open(filename, 'rb') as stream:
    db.myfile.insert(image=db.myfile.image.store(stream, filename))
```

It is also possible to insert a file in a simpler way and have the insert method call `store` automatically:

```
with open(filename, 'rb') as stream:
    db.myfile.insert(image=stream)
```

In this case the filename is obtained from the stream object if available.

The `store` method of the upload field object takes a file stream and a filename. It uses the filename to

determine the extension (type) of the file, creates a new temp name for the file (according to py4web upload mechanism) and loads the file content in this new temp file (under the uploads folder unless specified otherwise). It returns the new temp name, which is then stored in the `image` field of the `db.myfile` table.

Note, if the file is to be stored in an associated blob field rather than the file system, the `store` method will not insert the file in the blob field (because `store` is called before the insert), so the file must be explicitly inserted into the blob field:

```
db.define_table('myfile',
                Field('image', 'upload', uploadfield='image_file'),
                Field('image_file', 'blob'))
with open(filename, 'rb') as stream:
    db.myfile.insert(image=db.myfile.image.store(stream, filename),
                     image_file=stream.read())
```

The `retrieve` method does the opposite of `store`.

When uploaded files are stored on filesystem (as in the case of a plain `Field('image', 'upload')`) the code:

```
row = db(db.myfile).select().first()
(filename, fullname) = db.myfile.image.retrieve(row.image, nameonly=True)
```

retrieves the original file name (filename) as seen by the user at upload time and the name of stored file (fullname, with path relative to application folder). While in general the call:

```
(filename, stream) = db.myfile.image.retrieve(row.image)
```

retrieves the original file name (filename) and a file-like object ready to access uploaded file data (stream).

> Notice that the stream returned by `retrieve` is a real file object in the case that uploaded files are stored on filesystem. In that case remember to close the file when you are done, calling `stream.close()`.

Here is an example of safe usage of `retrieve`:

```
from contextlib import closing
import shutil
row = db(db.myfile).select().first()
(filename, stream) = db.myfile.image.retrieve(row.image)
with closing(stream) as src, closing(open(filename, 'wb')) as dest:
    shutil.copyfileobj(src, dest)
```

## 6.16 Query, Set, Rows

Let's consider again the table defined (and dropped) previously and insert three records:

```
>>> db.define_table('person', Field('name'))
<Table person (id, name)>
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
>>> db.person.insert(name="Carl")
3
```

You can store the table in a variable. For example, with variable `person`, you could do:

```
>>> person = db.person
```

You can also store a field in a variable such as `name`. For example, you could also do:

```
>>> name = person.name
```

You can even build a query (using operators like ==, !=, <, >, <=, >=, like, belongs) and store the query in a variable `q` such as in:

```
>>> q = name == 'Alex'
```

When you call `db` with a query, you define a set of records. You can store it in a variable `s` and write:

```
>>> s = db(q)
```

Notice that no database query has been performed so far. DAL + Query simply define a set of records in this db that match the query. py4web determines from the query which table (or tables) are involved and, in fact, there is no need to specify that.

## 6.17 select

Given a Set, `s`, you can fetch the records with the command `select`:

```
>>> rows = s.select()
```

It returns an iterable object of class `pydal.objects.Rows` whose elements are Row objects. `pydal.objects.Row` objects act like dictionaries, but their elements can also be accessed as attributes, like `gluon.storage.Storage`.The former differ from the latter because its values are read-only.

The Rows object allows looping over the result of the select and printing the selected field values for each row:

```
>>> for row in rows:
...     print row.id, row.name
...
1 Alex
```

You can do all the steps in one statement:

```
>>> for row in db(db.person.name == 'Alex').select():
...     print row.name
...
Alex
```

The select command can take arguments. All unnamed arguments are interpreted as the names of the fields that you want to fetch. For example, you can be explicit on fetching field "id" and field "name":

```
>>> for row in db().select(db.person.id, db.person.name):
...     print row.name
...
Alex
Bob
Carl
```

The table attribute ALL allows you to specify all fields:

```
>>> for row in db().select(db.person.ALL):
...     print row.id, row.name
...
1 Alex
2 Bob
3 Carl
```

Notice that there is no query string passed to db. py4web understands that if you want all fields of the table person without additional information then you want all records of the table person.

An equivalent alternative syntax is the following:

```
>>> for row in db(db.person).select():
...     print row.id, row.name
...
1 Alex
2 Bob
3 Carl
```

and py4web understands that if you ask for all records of the table person without additional information, then you want all the fields of table person.

Given one row

```
>>> row = rows[0]
```

you can extract its values using multiple equivalent expressions:

```
>>> row.name
Alex
>>> row['name']
Alex
>>> row('person.name')
Alex
```

The latter syntax is particularly handy when selecting en expression instead of a column. We will show this later.

You can also do

```
rows.compact = False
```

to disable the notation

```
rows[i].name
```

and enable, instead, the less compact notation:

```
rows[i].person.name
```

Yes this is unusual and rarely needed.

Row objects also have two important methods:

```
row.delete_record()
```

and

```
row.update_record(name="new value")
```

### 6.17.1 Using an iterator-based select for lower memory use

Python "iterators" are a type of "lazy-evaluation". They 'feed' data one step at time; traditional Python loops create the entire set of data in memory before looping.

The traditional use of select is:

```
for row in db(db.table).select():
    ...
```

but for large numbers of rows, using an iterator-based alternative has dramatically lower memory use:

```
for row in db(db.table).iterselect():
    ...
```

Testing shows this is around 10% faster as well, even on machines with large RAM.

### 6.17.2 Rendering rows using represent

You may wish to rewrite rows returned by select to take advantage of formatting information contained in the represents setting of the fields.

```
rows = db(query).select()
repr_row = rows.render(0)
```

If you don't specify an index, you get a generator to iterate over all the rows:

```
for row in rows.render():
    print row.myfield
```

Can also be applied to slices:

```
for row in rows[0:10].render():
    print row.myfield
```

If you only want to transform selected fields via their "represent" attribute, you can list them in the "fields" argument:

```
repr_row = row.render(0, fields=[db.mytable.myfield])
```

Note, it returns a transformed copy of the original Row, so there's no update_record (which you wouldn't want anyway) or delete_record.

### 6.17.3 Shortcuts

The DAL supports various code-simplifying shortcuts. In particular:

```
myrecord = db.mytable[id]
```

returns the record with the given `id` if it exists. If the `id` does not exist, it returns `None`. The above statement is equivalent to

```
myrecord = db(db.mytable.id == id).select().first()
```

You can delete records by id:

```
del db.mytable[id]
```

and this is equivalent to

```
db(db.mytable.id == id).delete()
```

and deletes the record with the given `id`, if it exists.

Note: this delete shortcut syntax does not currently work if *versioning* is activated

You can insert records:

```
db.mytable[None] = dict(myfield='somevalue')
```

It is equivalent to

```
db.mytable.insert(myfield='somevalue')
```

and it creates a new record with field values specified by the dictionary on the right hand side.

Note: insert shortcut was previously `db.table[0] = ...`. It has changed in PyDAL 19.02 to permit normal usage of id 0.

You can update records:

```
db.mytable[id] = dict(myfield='somevalue')
```

which is equivalent to

```
db(db.mytable.id == id).update(myfield='somevalue')
```

and it updates an existing record with field values specified by the dictionary on the right hand side.

### 6.17.4 Fetching a `Row`

Yet another convenient syntax is the following:

```
record = db.mytable(id)
record = db.mytable(db.mytable.id == id)
record = db.mytable(id, myfield='somevalue')
```

Apparently similar to `db.mytable[id]` the above syntax is more flexible and safer. First of all it checks whether `id` is an int (or `str(id)` is an int) and returns `None` if not (it never raises an exception). It also allows to specify multiple conditions that the record must meet. If they are not met, it also returns `None`.

### 6.17.5 Recursive `selects`

Consider the previous table person and a new table "thing" referencing a "person":

```
db.define_table('thing',
                Field('name'),
                Field('owner_id', 'reference person'))
```

and a simple select from this table:

```
things = db(db.thing).select()
```

which is equivalent to

```
things = db(db.thing._id != None).select()
```

where `_id` is a reference to the primary key of the table. Normally `db.thing._id` is the same as `db.thing.id` and we will assume that in most of this book.

For each Row of things it is possible to fetch not just fields from the selected table (thing) but also from linked tables (recursively):

```
for thing in things:
    print thing.name, thing.owner_id.name
```

Here `thing.owner_id.name` requires one database select for each thing in things and it is therefore inefficient. We suggest using joins whenever possible instead of recursive selects, nevertheless this is convenient and practical when accessing individual records.

You can also do it backwards, by selecting the things referenced by a person:

```
person = db.person(id)
for thing in person.thing.select(orderby=db.thing.name):
    print person.name, 'owns', thing.name
```

In this last expression `person.thing` is a shortcut for

```
db(db.thing.owner_id == person.id)
```

i.e. the Set of `things` referenced by the current `person`. This syntax breaks down if the referencing table has multiple references to the referenced table. In this case one needs to be more explicit and use a full Query.

### 6.17.6 orderby, groupby, limitby, distinct, having, orderby_on_limitby, join, left, cache

The `select` command takes a number of optional arguments.

#### orderby

You can fetch the records sorted by name:

```
>>> for row in db().select(db.person.ALL, orderby=db.person.name):
...     print row.name
...
Alex
Bob
Carl
```

You can fetch the records sorted by name in reverse order (notice the tilde):

```
>>> for row in db().select(db.person.ALL, orderby=~db.person.name):
...     print row.name
...
Carl
Bob
Alex
```

You can have the fetched records appear in random order:

```
>>> for row in db().select(db.person.ALL, orderby='<random>'):
...     print row.name
```

```
...
Carl
Alex
Bob
```

The use of `orderby='<random>'` is not supported on Google NoSQL. However, to overcome this limit, sorting can be accomplished on selected rows:

```
import random
rows = db(...).select().sort(lambda row: random.random())
```

You can sort the records according to multiple fields by concatenating them with a "|":

```
>>> for row in db().select(db.person.name, orderby=db.person.name|db.person.id):
...     print row.name
...
Alex
Bob
Carl
```

#### groupby, having

Using `groupby` together with `orderby`, you can group records with the same value for the specified field (this is back-end specific, and is not on the Google NoSQL):

```
>>> for row in db().select(db.person.ALL,
...                        orderby=db.person.name,
...                        groupby=db.person.name):
...     print row.name
...
Alex
Bob
Carl
```

You can use `having` in conjunction with `groupby` to group conditionally (only those `having` the condition are grouped).

```
>>> print db(query1).select(db.person.ALL, groupby=db.person.name, having=query2)
```

Notice that query1 filters records to be displayed, query2 filters records to be grouped.

#### distinct

With the argument `distinct=True`, you can specify that you only want to select distinct records. This has the same effect as grouping using all specified fields except that it does not require sorting. When using distinct it is important not to select ALL fields, and in particular not to select the "id" field, else all records will always be distinct.

Here is an example:

```
>>> for row in db().select(db.person.name, distinct=True):
...     print row.name
...
Alex
Bob
Carl
```

Notice that `distinct` can also be an expression, for example:

```
>>> for row in db().select(db.person.name, distinct=db.person.name):
...     print row.name
...
Alex
Bob
Carl
```

#### limitby

With `limitby=(min, max)`, you can select a subset of the records from offset=min to but not including offset=max. In the next example we select the first two records starting at zero:

```
>>> for row in db().select(db.person.ALL, limitby=(0, 2)):
...     print row.name
...
Alex
Bob
```

#### orderby_on_limitby

Note that the DAL defaults to implicitly adding an orderby when using a limitby. This ensures the same query returns the same results each time, important for pagination. But it can cause performance problems. use `orderby_on_limitby = False` to change this (this defaults to True).

#### join, left

These are involved in managing *one to many relations* . They are described in *Inner join* and *Left outer join* sections respectively.

#### cache, cacheable

An example use which gives much faster selects is: `rows = db(query).select(cache=(cache.ram, 3600), cacheable=True)`:python Look at *Caching selects* section in this chapter, to understand what the trade-offs are.

### 6.17.7 Logical operators

Queries can be combined using the binary AND operator "`&`":

```
>>> rows = db((db.person.name=='Alex') & (db.person.id > 3)).select()
>>> for row in rows: print row.id, row.name
>>> len(rows)
0
```

and the binary OR operator "`|`":

```
>>> rows = db((db.person.name == 'Alex') | (db.person.id > 3)).select()
>>> for row in rows: print row.id, row.name
1 Alex
```

You can negate a sub-query inverting its operator:

```
>>> rows = db((db.person.name != 'Alex') | (db.person.id > 3)).select()
>>> for row in rows: print row.id, row.name
2 Bob
3 Carl
```

or by explicit negation with the "`~`" unary operator:

```
>>> rows = db(~(db.person.name == 'Alex') | (db.person.id > 3)).select()
>>> for row in rows: print row.id, row.name
2 Bob
3 Carl
```

Due to Python restrictions in overloading "and" and "or" operators, these cannot be used in forming queries. The binary operators "&" and "|" must be used instead. Note that these operators (unlike "and" and "or") have higher precedence than comparison operators, so the "extra" parentheses in the above examples are mandatory. Similarly, the unary operator "~" has higher precedence than comparison operators, so ~-negated comparisons must also be parenthesized.

It is also possible to build queries using in-place logical operators:

```
>>> query = db.person.name != 'Alex'
>>> query &= db.person.id > 3
>>> query |= db.person.name == 'John'
```

### 6.17.8 count, isempty, delete, update

You can count records in a set:

```
>>> db(db.person.name != 'William').count()
3
```

Notice that `count` takes an optional `distinct` argument which defaults to False, and it works very much like the same argument for `select`. `count` has also a `cache` argument that works very much like the equivalent argument of the `select` method.

Sometimes you may need to check if a table is empty. A more efficient way than counting is using the `isempty` method:

```
>>> db(db.person).isempty()
False
```

You can delete records in a set:

```
>>> db(db.person.id > 3).delete()
0
```

The `delete` method returns the number of records that were deleted.

And you can update all records in a set by passing named arguments corresponding to the fields that need to be updated:

```
>>> db(db.person.id > 2).update(name='Ken')
1
```

The `update` method returns the number of records that were updated.

### 6.17.9 Expressions

The value assigned an update statement can be an expression. For example consider this model

```
db.define_table('person',
                Field('name'),
                Field('visits', 'integer', default=0))
```

```
db(db.person.name == 'Massimo').update(visits = db.person.visits + 1)
```

The values used in queries can also be expressions

```
db.define_table('person',
                Field('name'),
                Field('visits', 'integer', default=0),
                Field('clicks', 'integer', default=0))

db(db.person.visits == db.person.clicks + 1).delete()
```

### 6.17.10 case

An expression can contain a case clause for example:

```
>>> condition = db.person.name.startswith('B')
>>> yes_or_no = condition.case('Yes', 'No')
>>> for row in db().select(db.person.name, yes_or_no):
...     print row.person.name, row[yes_or_no]  # could be row(yes_or_no) too
...
Alex No
Bob Yes
Ken No
```

### 6.17.11 update_record

py4web also allows updating a single record that is already in memory using `update_record`

```
>>> row = db(db.person.id == 2).select().first()
>>> row.update_record(name='Curt')
<Row {'id': 2L, 'name': 'Curt'}>
```

`update_record` should not be confused with

```
>>> row.update(name='Curt')
```

because for a single row, the method `update` updates the row object but not the database record, as in the case of `update_record`.

It is also possible to change the attributes of a row (one at a time) and then call `update_record()` without arguments to save the changes:

```
>>> row = db(db.person.id > 2).select().first()
>>> row.name = 'Philip'
>>> row.update_record() # saves above change
<Row {'id': 3L, 'name': 'Philip'}>
```

> Note, you should avoid using `row.update_record()` with no arguments when the `row` object contains fields that have an `update` attribute (e.g., `Field('modified_on', update=request.now)`). Calling `row.update_record()` will retain *all* of the existing values in the `row` object, so any fields with `update` attributes will have no effect in this case. Be particularly mindful of this with tables that include `auth.signature`.

The `update_record` method is available only if the table's `id` field is included in the select, and `cacheable` is not set to `True`.

### 6.17.12 Inserting and updating from a dictionary

A common issue consists of needing to insert or update records in a table where the name of the table, the field to be updated, and the value for the field are all stored in variables. For example: `tablename`, `fieldname`, and `value`.

The insert can be done using the following syntax:

```
db[tablename].insert(**{fieldname:value})
```

The update of record with given id can be done with:

```
db(db[tablename]._id == id).update(**{fieldname:value})
```

Notice we used `table._id` instead of `table.id`. In this way the query works even for tables with a primary key field with type other than "id".

### 6.17.13 `first` and `last`

Given a Rows object containing records:

```
rows = db(query).select()
first_row = rows.first()
last_row = rows.last()
```

are equivalent to

```
first_row = rows[0] if len(rows) else None
last_row = rows[-1] if len(rows) else None
```

Notice, `first()` and `last()` allow you to obtain obviously the first and last record present in your query, but this won't mean that these records are going to be the first or last inserted records. In case you want the first or last record inputted in a given table don't forget to use `orderby=db.table_name.id`. If you forget you will only get the first and last record returned by your query which are often in a random order determined by the backend query optimiser.

### 6.17.14 `as_dict` and `as_list`

A Row object can be serialized into a regular dictionary using the `as_dict()` method and a Rows object can be serialized into a list of dictionaries using the `as_list()` method. Here are some examples:

```
rows = db(query).select()
rows_list = rows.as_list()
first_row_dict = rows.first().as_dict()
```

These methods are convenient for passing Rows to generic views and or to store Rows in sessions (since Rows objects themselves cannot be serialized since contain a reference to an open DB connection):

```
rows = db(query).select()
session.rows = rows  # not allowed!
session.rows = rows.as_list()  # allowed!
```

### 6.17.15 Combining rows

Rows objects can be combined at the Python level. Here we assume:

```
>>> print rows1
person.name
Max
Tim

>>> print rows2
person.name
John
Tim
```

You can do union of the records in two sets of rows:

```
>>> rows3 = rows1 + rows2
>>> print rows3
person.name
Max
Tim
John
Tim
```

You can do union of the records removing duplicates:

```
>>> rows3 = rows1 | rows2
>>> print rows3
person.name
Max
Tim
John
```

You can do intersection of the records in two sets of rows:

```
>>> rows3 = rows1 & rows2
>>> print rows3
person.name
Tim
```

### 6.17.16 `find`, `exclude`, `sort`

Some times you need to perform two selects and one contains a subset of a previous select. In this case it is pointless to access the database again. The `find`, `exclude` and `sort` objects allow you to manipulate a Rows object and generate another one without accessing the database. More specifically: - `find` returns a new set of Rows filtered by a condition and leaves the original unchanged. - `exclude` returns a new set of Rows filtered by a condition and removes them from the original Rows. - `sort` returns a new set of Rows sorted by a condition and leaves the original unchanged.

All these methods take a single argument, a function that acts on each individual row.

Here is an example of usage:

```
>>> db.define_table('person', Field('name'))
<Table person (id, name)>
>>> db.person.insert(name='John')
1
>>> db.person.insert(name='Max')
2
>>> db.person.insert(name='Alex')
3
>>> rows = db(db.person).select()
```

```
>>> for row in rows.find(lambda row: row.name[0]=='M'):
...     print row.name
...
Max
>>> len(rows)
3
>>> for row in rows.exclude(lambda row: row.name[0]=='M'):
...     print row.name
...
Max
>>> len(rows)
2
>>> for row in rows.sort(lambda row: row.name):
...     print row.name
...
Alex
John
```

They can be combined:

```
>>> rows = db(db.person).select()
>>> rows = rows.find(lambda row: 'x' in row.name).sort(lambda row: row.name)
>>> for row in rows:
...     print row.name
...
Alex
Max
```

Sort takes an optional argument `reverse=True` with the obvious meaning.

The `find` method has an optional `limitby` argument with the same syntax and functionality as the Set `select` method.

## 6.18 Other methods

### 6.18.1 update_or_insert

Some times you need to perform an insert only if there is no record with the same values as those being inserted. This can be done with

```
db.define_table('person',
                Field('name'),
                Field('birthplace'))

db.person.update_or_insert(name='John', birthplace='Chicago')
```

The record will be inserted only if there is no other user called John born in Chicago.

You can specify which values to use as a key to determine if the record exists. For example:

```
db.person.update_or_insert(db.person.name == 'John',
                           name='John',
                           birthplace='Chicago')
```

and if there is John his birthplace will be updated else a new record will be created.

The selection criteria in the example above is a single field. It can also be a query, such as

```
db.person.update_or_insert((db.person.name == 'John') & (db.person.birthplace ==
'Chicago'),
                           name='John',
                           birthplace='Chicago',
                           pet='Rover')
```

### 6.18.2 validate_and_insert, validate_and_update

The function

```
ret = db.mytable.validate_and_insert(field='value')
```

works very much like

```
id = db.mytable.insert(field='value')
```

except that it calls the validators for the fields before performing the insert and bails out if the validation does not pass. If validation does not pass the errors can be found in `ret.errors`. `ret.errors` holds a key-value mapping where each key is the field name whose validation failed, and the value of the key is the result from the validation error (much like `form.errors`). If it passes, the id of the new record is in `ret.id`. Mind that normally validation is done by the form processing logic so this function is rarely needed.

Similarly

```
ret = db(query).validate_and_update(field='value')
```

works very much the same as

```
num = db(query).update(field='value')
```

except that it calls the validators for the fields before performing the update. Notice that it only works if query involves a single table. The number of updated records can be found in `ret.updated` and errors will be in `ret.errors`.

## 6.19 Computed fields

DAL fields may have a `compute` attribute. This must be a function (or lambda) that takes a Row object and returns a value for the field. When a new record is modified, including both insertions and updates, if a value for the field is not provided, py4web tries to compute from the other field values using the `compute` function. Here is an example:

```
>>> db.define_table('item',
...                 Field('unit_price', 'double'),
...                 Field('quantity', 'integer'),
...                 Field('total_price',
...                       compute=lambda r: r['unit_price'] * r['quantity']))
<Table item (id, unit_price, quantity, total_price)>
>>> rid = db.item.insert(unit_price=1.99, quantity=5)
>>> db.item[rid]
<Row {'total_price': '9.95', 'unit_price': 1.99, 'id': 1L, 'quantity': 5L}>
```

Notice that the computed value is stored in the db and it is not computed on retrieval, as in the case of virtual fields, described next. Two typical applications of computed fields are: - in wiki applications, to

store the processed input wiki text as HTML, to avoid re-processing on every request - for searching, to compute normalized values for a field, to be used for searching.

Computed fields are evaluated in the order in which they are defined in the table definition. A computed field can refer to previously defined computed fields (new after v 2.5.1)

## 6.20 Virtual fields

Virtual fields are also computed fields (as in the previous subsection) but they differ from those because they are *virtual* in the sense that they are not stored in the db and they are computed each time records are extracted from the database. They can be used to simplify the user's code without using additional storage but they cannot be used for searching.

### 6.20.1 New style virtual fields (experimental)

py4web provides a new and easier way to define virtual fields and lazy virtual fields. This section is marked experimental because the APIs may still change a little from what is described here.

Here we will consider the same example as in the previous subsection. In particular we consider the following model:

```
db.define_table('item',
                Field('unit_price', 'double'),
                Field('quantity', 'integer'))
```

One can define a `total_price` virtual field as

```
db.item.total_price = Field.Virtual(lambda row: row.item.unit_price *
row.item.quantity)
```

i.e. by simply defining a new field `total_price` to be a `Field.Virtual`. The only argument of the constructor is a function that takes a row and returns the computed values.

A virtual field defined as the one above is automatically computed for all records when the records are selected:

```
for row in db(db.item).select():
    print row.total_price
```

It is also possible to define method fields which are calculated on-demand, when called. For example:

```
db.item.discounted_total = \\
    Field.Method(lambda row, discount=0.0:
                 row.item.unit_price * row.item.quantity * (100.0 - discount / 100))
```

In this case `row.discounted_total` is not a value but a function. The function takes the same arguments as the function passed to the `Method` constructor except for `row` which is implicit (think of it as `self` for objects).

The lazy field in the example above allows one to compute the total price for each `item`:

```
for row in db(db.item).select(): print row.discounted_total()
```

And it also allows to pass an optional `discount` percentage (say 15%):

```
for row in db(db.item).select(): print row.discounted_total(15)
```

Virtual and Method fields can also be defined in place when a table is defined:

```
db.define_table('item',
                Field('unit_price', 'double'),
                Field('quantity', 'integer'),
                Field.Virtual('total_price', lambda row: ...),
                Field.Method('discounted_total', lambda row, discount=0.0: ...))
```

Mind that virtual fields do not have the same attributes as regular fields (length, default, required, etc). They do not appear in the list of `db.table.fields` and in older versions of py4web they require a special approach to display in SQLFORM.grid and SQLFORM.smartgrid. See the discussion on grids and virtual fields in *Chapter 7 ../07* .

### 6.20.2 Old style virtual fields

In order to define one or more virtual fields, you can also define a container class, instantiate it and link it to a table or to a select. For example, consider the following table:

```
db.define_table('item',
                Field('unit_price', 'double'),
                Field('quantity', 'integer'))
```

One can define a `total_price` virtual field as

```
class MyVirtualFields(object):
    def total_price(self):
        return self.item.unit_price * self.item.quantity

db.item.virtualfields.append(MyVirtualFields())
```

Notice that each method of the class that takes a single argument (self) is a new virtual field. `self` refers to each one row of the select. Field values are referred by full path as in `self.item.unit_price`. The table is linked to the virtual fields by appending an instance of the class to the table's `virtualfields` attribute.

Virtual fields can also access recursive fields as in

```
db.define_table('item',
                Field('unit_price', 'double'))

db.define_table('order_item',
                Field('item', 'reference item'),
                Field('quantity', 'integer'))

class MyVirtualFields(object):
    def total_price(self):
        return self.order_item.item.unit_price * self.order_item.quantity

db.order_item.virtualfields.append(MyVirtualFields())
```

Notice the recursive field access `self.order_item.item.unit_price` where `self` is the looping record.

They can also act on the result of a JOIN

```
rows = db(db.order_item.item == db.item.id).select()

class MyVirtualFields(object):
```

```
    def total_price(self):
        return self.item.unit_price * self.order_item.quantity

rows.setvirtualfields(order_item=MyVirtualFields())

for row in rows:
    print row.order_item.total_price
```

Notice how in this case the syntax is different. The virtual field accesses both `self.item.unit_price` and `self.order_item.quantity` which belong to the join select. The virtual field is attached to the rows of the table using the `setvirtualfields` method of the rows object. This method takes an arbitrary number of named arguments and can be used to set multiple virtual fields, defined in multiple classes, and attach them to multiple tables:

```
class MyVirtualFields1(object):
    def discounted_unit_price(self):
        return self.item.unit_price * 0.90

class MyVirtualFields2(object):
    def total_price(self):
        return self.item.unit_price * self.order_item.quantity
    def discounted_total_price(self):
        return self.item.discounted_unit_price * self.order_item.quantity

rows.setvirtualfields(item=MyVirtualFields1(),
                      order_item=MyVirtualFields2())

for row in rows:
    print row.order_item.discounted_total_price
```

Virtual fields can be *lazy*; all they need to do is return a function and access it by calling the function:

```
db.define_table('item',
                Field('unit_price', 'double'),
                Field('quantity', 'integer'))

class MyVirtualFields(object):
    def lazy_total_price(self):
        def lazy(self=self):
            return self.item.unit_price * self.item.quantity
        return lazy

db.item.virtualfields.append(MyVirtualFields())

for item in db(db.item).select():
    print item.lazy_total_price()
```

or shorter using a lambda function:

```
class MyVirtualFields(object):
    def lazy_total_price(self):
        return lambda self=self: self.item.unit_price * self.item.quantity
```

## 6.21 One to many relation

To illustrate how to implement one to many relations with the DAL, define another table "thing" that

refers to the table "person" which we redefine here:

```
>>> db.define_table('person',
...                  Field('name'))
<Table person (id, name)>
>>> db.person.insert(name='Alex')
1
>>> db.person.insert(name='Bob')
2
>>> db.person.insert(name='Carl')
3
>>> db.define_table('thing',
...                  Field('name'),
...                  Field('owner_id', 'reference person'))
<Table thing (id, name, owner_id)>
```

Table "thing" has two fields, the name of the thing and the owner of the thing. The "owner_id" field is a reference field, it is intended that the field reference the other table by its id. A reference type can be specified in two equivalent ways, either: `Field('owner_id', 'reference person')`:python or: `Field('owner_id', db.person)`:python

The latter is always converted to the former. They are equivalent except in the case of lazy tables, self references or other types of cyclic references where the former notation is the only allowed notation.

Now, insert three things, two owned by Alex and one by Bob:

```
>>> db.thing.insert(name='Boat', owner_id=1)
1
>>> db.thing.insert(name='Chair', owner_id=1)
2
>>> db.thing.insert(name='Shoes', owner_id=2)
3
```

You can select as you did for any other table:

```
>>> for row in db(db.thing.owner_id == 1).select():
...     print row.name
...
Boat
Chair
```

Because a thing has a reference to a person, a person can have many things, so a record of table person now acquires a new attribute thing, which is a Set, that defines the things of that person. This allows looping over all persons and fetching their things easily:

```
>>> for person in db().select(db.person.ALL):
...     print person.name
...     for thing in person.thing.select():
...         print '    ', thing.name
...
Alex
     Boat
     Chair
Bob
     Shoes
Carl
```

### 6.21.1 Inner joins

Another way to achieve a similar result is by using a join, specifically an INNER JOIN. py4web performs joins automatically and transparently when the query links two or more tables as in the following example:

```
>>> rows = db(db.person.id == db.thing.owner_id).select()
>>> for row in rows:
...     print row.person.name, 'has', row.thing.name
...
Alex has Boat
Alex has Chair
Bob has Shoes
```

Observe that py4web did a join, so the rows now contain two records, one from each table, linked together. Because the two records may have fields with conflicting names, you need to specify the table when extracting a field value from a row. This means that while before you could do:

```
row.name
```

and it was obvious whether this was the name of a person or a thing, in the result of a join you have to be more explicit and say:

```
row.person.name
```

or:

```
row.thing.name
```

There is an alternative syntax for INNER JOINS:

```
>>> rows = db(db.person).select(join=db.thing.on(db.person.id == db.thing.owner_id))
>>> for row in rows:
...     print row.person.name, 'has', row.thing.name
...
Alex has Boat
Alex has Chair
Bob has Shoes
```

While the output is the same, the generated SQL in the two cases can be different. The latter syntax removes possible ambiguities when the same table is joined twice and aliased:

```
db.define_table('thing',
                Field('name'),
                Field('owner_id1', 'reference person'),
                Field('owner_id2', 'reference person'))

rows = db(db.person).select(
        join=[db.person.with_alias('owner_id1').on(db.person.id ==
db.thing.owner_id1),
              db.person.with_alias('owner_id2').on(db.person.id ==
db.thing.owner_id2)])
```

The value of `join` can be list of `db.table.on(...)` to join.

### 6.21.2 Left outer join

Notice that Carl did not appear in the list above because he has no things. If you intend to select on persons (whether they have things or not) and their things (if they have any), then you need to perform a LEFT OUTER JOIN. This is done using the argument "left" of the select. Here is an example:

```
>>> rows = db().select(db.person.ALL, db.thing.ALL,
...                    left=db.thing.on(db.person.id == db.thing.owner_id))
>>> for row in rows:
...     print row.person.name, 'has', row.thing.name
...
Alex has Boat
Alex has Chair
Bob has Shoes
Carl has None
```

where:

```
left = db.thing.on(...)
```

does the left join query. Here the argument of `db.thing.on` is the condition required for the join (the same used above for the inner join). In the case of a left join, it is necessary to be explicit about which fields to select.

Multiple left joins can be combined by passing a list or tuple of `db.mytable.on(...)` to the `left` parameter.

### 6.21.3 Grouping and counting

When doing joins, sometimes you want to group rows according to certain criteria and count them. For example, count the number of things owned by every person. py4web allows this as well. First, you need a count operator. Second, you want to join the person table with the thing table by owner. Third, you want to select all rows (person + thing), group them by person, and count them while grouping:

```
>>> count = db.person.id.count()
>>> for row in db(db.person.id == db.thing.owner_id
...               ).select(db.person.name, count, groupby=db.person.name):
...     print row.person.name, row[count]
...
Alex 2
Bob 1
```

Notice the `count` operator (which is built-in) is used as a field. The only issue here is in how to retrieve the information. Each row clearly contains a person and the count, but the count is not a field of a person nor is it a table. So where does it go? It goes into the storage object representing the record with a key equal to the query expression itself.

The `count` method of the Field object has an optional `distinct` argument. When set to `True` it specifies that only distinct values of the field in question are to be counted.

## 6.22 Many to many

In the previous examples, we allowed a thing to have one owner but one person could have many things. What if Boat was owned by Alex and Curt? This requires a many-to-many relation, and it is realized via an intermediate table that links a person to a thing via an ownership relation.

Here is how to do it:

```
>>> db.define_table('person',
...                  Field('name'))
<Table person (id, name)>
>>> db.person.bulk_insert([dict(name='Alex'), dict(name='Bob'), dict(name='Carl')])
[1, 2, 3]
>>> db.define_table('thing',
...                  Field('name'))
<Table thing (id, name)>
>>> db.thing.bulk_insert([dict(name='Boat'), dict(name='Chair'), dict(name='Shoes')])
[1, 2, 3]
>>> db.define_table('ownership',
...                  Field('person', 'reference person'),
...                  Field('thing', 'reference thing'))
<Table ownership (id, person, thing)>
```

the existing ownership relationship can now be rewritten as:

```
>>> db.ownership.insert(person=1, thing=1)  # Alex owns Boat
1
>>> db.ownership.insert(person=1, thing=2)  # Alex owns Chair
2
>>> db.ownership.insert(person=2, thing=3)  # Bob owns Shoes
3
```

Now you can add the new relation that Curt co-owns Boat:

```
>>> db.ownership.insert(person=3, thing=1)  # Curt owns Boat too
4
```

Because you now have a three-way relation between tables, it may be convenient to define a new set on which to perform operations:

```
>>> persons_and_things = db((db.person.id == db.ownership.person) &
...                         (db.thing.id == db.ownership.thing))
```

Now it is easy to select all persons and their things from the new Set:

```
>>> for row in persons_and_things.select():
...     print row.person.name, 'has', row.thing.name
...
Alex has Boat
Alex has Chair
Bob has Shoes
Curt has Boat
```

Similarly, you can search for all things owned by Alex:

```
>>> for row in persons_and_things(db.person.name == 'Alex').select():
...     print row.thing.name
...
Boat
Chair
```

and all owners of Boat:

```
>>> for row in persons_and_things(db.thing.name == 'Boat').select():
```

```
...     print row.person.name
...
Alex
Curt
```

A lighter alternative to many-to-many relations is tagging, you can found an example of this in the next section. Tagging works even on database backends that do not support JOINs like the Google App Engine NoSQL.

## 6.23 Tagging records

Tags allows to add or find properties attached to records in your database.

```
from pydal import DAL, Field
from py4web.utils.tags import Tags

db = DAL("sqlite:memory")
db.define_table("thing", Field("name"))
properties = Tags(db.thing)
id1 = db.thing.insert(name="chair")
id2 = db.thing.insert(name="table")
properties.add(id1, "color/red")
properties.add(id1, "style/modern")
properties.add(id2, "color/green")
properties.add(id2, "material/wood")

self.assertTrue(properties.get(id1), ["color/red", "style/modern"])
self.assertTrue(properties.get(id2), ["color/green", "material/wood"])

rows = db(properties.find(["style/modern"])).select()
self.assertTrue(rows.first().id, id1)

rows = db(properties.find(["material/wood"])).select()
self.assertTrue(rows.first().id, id1)

rows = db(properties.find(["color"])).select()
self.assertTrue(len(rows), 2)
```

It is internally implemented as a table with name: *tags*, which in this example would be db.thing_tags_default, because no path was specified on the Tags(table, path="default") constructor

The `find` method is doing a search by `startswith` of the path passed as parameter. Then find(["color"]) would return id1 and id2 because both records have tags starting with "color" You can find some examples of record's tagging in chapter 11, as py4web uses tags as a flexible mechanism to manage permissions.

## 6.24 `list:<type>` and `contains`

py4web provides the following special field types:

```
list:string
list:integer
list:reference <table>
```

They can contain lists of strings, of integers and of references respectively.

On Google App Engine NoSQL `list:string` is mapped into `StringListProperty`, the other two are mapped into `ListProperty(int)`. On relational databases they are mapped into text fields which contain the list of items separated by `|`. For example `[1, 2, 3]` is mapped into `|1|2|3|`.

For lists of string the items are escaped so that any `|` in the item is replaced by a `||`. Anyway this is an internal representation and it is transparent to the user.

You can use `list:string`, for example, in the following way:

```
>>> db.define_table('product',
...                  Field('name'),
...                  Field('colors', 'list:string'))
<Table product (id, name, colors)>
>>> db.product.colors.requires = IS_IN_SET(('red', 'blue', 'green'))
>>> db.product.insert(name='Toy Car', colors=['red', 'green'])
1
>>> products = db(db.product.colors.contains('red')).select()
>>> for item in products:
...     print item.name, item.colors
...
Toy Car ['red', 'green']
```

`list:integer` works in the same way but the items must be integers.

As usual the requirements are enforced at the level of forms, not at the level of `insert`.

> For `list:<type>` fields the `contains(value)` operator maps into a non trivial query that checks for lists containing the `value`. The `contains` operator also works for regular `string` and `text` fields and it maps into a `LIKE '%value%'`.

The `list:reference` and the `contains(value)` operator are particularly useful to de-normalize many-to-many relations. Here is an example:

```
>>> db.define_table('tag',
...                  Field('name'),
...                  format='%(name)s')
<Table tag (id, name)>
>>> db.define_table('product',
...                  Field('name'),
...                  Field('tags', 'list:reference tag'))
<Table product (id, name, tags)>
>>> a = db.tag.insert(name='red')
>>> b = db.tag.insert(name='green')
>>> c = db.tag.insert(name='blue')
>>> db.product.insert(name='Toy Car', tags=[a, b, c])
1
>>> products = db(db.product.tags.contains(b)).select()
>>> for item in products:
...     print item.name, item.tags
...
Toy Car [1, 2, 3]
>>> for item in products:
...     print item.name, db.product.tags.represent(item.tags)
...
Toy Car red, green, blue
```

Notice that a `list:reference tag` field get a default constraint

```
requires = IS_IN_DB(db, db.tag._id, db.tag._format, multiple=True)
```

that produces a `SELECT/OPTION` multiple drop-box in forms.

Also notice that this field gets a default `represent` attribute which represents the list of references as a comma-separated list of formatted references. This is used in read `forms`.

> While `list:reference` has a default validator and a default representation, `list:integer` and `list:string` do not. So these two need an `IS_IN_SET` or an `IS_IN_DB` validator if you want to use them in forms.

## 6.25 Other operators

py4web has other operators that provide an API to access equivalent SQL operators. Let's define another table "log" to store security events, their event_time and severity, where the severity is an integer number.

```
>>> db.define_table('log', Field('event'),
...                         Field('event_time', 'datetime'),
...                         Field('severity', 'integer'))
<Table log (id, event, event_time, severity)>
```

As before, insert a few events, a "port scan", an "xss injection" and an "unauthorized login". For the sake of the example, you can log events with the same event_time but with different severities (1, 2, and 3 respectively).

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> db.log.insert(event='port scan', event_time=now, severity=1)
1
>>> db.log.insert(event='xss injection', event_time=now, severity=2)
2
>>> db.log.insert(event='unauthorized login', event_time=now, severity=3)
3
```

### 6.25.1 `like`, `ilike`, `regexp`, `startswith`, `endswith`, `contains`, `upper`, `lower`

Fields have a `like` operator that you can use to match strings:

```
>>> for row in db(db.log.event.like('port%')).select():
...     print row.event
...
port scan
```

Here "port%" indicates a string starting with "port". The percent sign character, "%", is a wild-card character that means "any sequence of characters".

The `like` operator maps to the LIKE word in ANSI-SQL. LIKE is case-sensitive in most databases, and depends on the collation of the database itself. The `like` method is hence case-sensitive but it can be made case-insensitive with

```
db.mytable.myfield.like('value', case_sensitive=False)
```

which is the same as using `ilike`

```
db.mytable.myfield.ilike('value')
```

py4web also provides some shortcuts:

```
db.mytable.myfield.startswith('value')
db.mytable.myfield.endswith('value')
db.mytable.myfield.contains('value')
```

which are roughly equivalent respectively to

```
db.mytable.myfield.like('value%')
db.mytable.myfield.like('%value')
db.mytable.myfield.like('%value%')
```

Remember that `contains` has a special meaning for `list:<type>` fields, as discussed in previous *list: and contains* section.

The `contains` method can also be passed a list of values and an optional boolean argument `all` to search for records that contain all values:

```
db.mytable.myfield.contains(['value1', 'value2'], all=True)
```

or any value from the list

```
db.mytable.myfield.contains(['value1', 'value2'], all=False)
```

There is a also a `regexp` method that works like the `like` method but allows regular expression syntax for the look-up expression. It is only supported by MySQL, Oracle, PostgreSQL, SQLite, and MongoDB (with different degree of support).

The `upper` and `lower` methods allow you to convert the value of the field to upper or lower case, and you can also combine them with the like operator:

```
>>> for row in db(db.log.event.upper().like('PORT%')).select():
...     print row.event
...
port scan
```

### 6.25.2 year, month, day, hour, minutes, seconds

The date and datetime fields have `day`, `month` and `year` methods. The datetime and time fields have `hour`, `minutes` and `seconds` methods. Here is an example:

```
>>> for row in db(db.log.event_time.year() > 2018).select():
...     print row.event
...
port scan
xss injection
unauthorized login
```

### 6.25.3 belongs

The SQL IN operator is realized via the `belongs` method which returns true when the field value belongs to the specified set (list or tuples):

```
>>> for row in db(db.log.severity.belongs((1, 2))).select():
...     print row.event
...
port scan
```

```
xss injection
```

The DAL also allows a nested select as the argument of the belongs operator. The only caveat is that the nested select has to be a `_select`, not a `select`, and only one field has to be selected explicitly, the one that defines the set.

```
>>> bad_days = db(db.log.severity == 3)._select(db.log.event_time)
>>> for row in db(db.log.event_time.belongs(bad_days)).select():
...     print row.severity, row.event
...
1 port scan
2 xss injection
3 unauthorized login
```

In those cases where a nested select is required and the look-up field is a reference we can also use a query as argument. For example:

```
db.define_table('person', Field('name'))
db.define_table('thing',
                Field('name'),
                Field('owner_id', 'reference person'))

db(db.thing.owner_id.belongs(db.person.name == 'Jonathan')).select()
```

In this case it is obvious that the nested select only needs the field referenced by the `db.thing.owner_id` field so we do not need the more verbose `_select` notation.

A nested select can also be used as insert/update value but in this case the syntax is different:

```
lazy = db(db.person.name == 'Jonathan').nested_select(db.person.id)

db(db.thing.id == 1).update(owner_id = lazy)
```

In this case `lazy` is a nested expression that computes the `id` of person "Jonathan". The two lines result in one single SQL query.

### 6.25.4 `sum`, `avg`, `min`, `max` and `len`

Previously, you have used the `count` operator to count records. Similarly, you can use the `sum` operator to add (sum) the values of a specific field from a group of records. As in the case of count, the result of a sum is retrieved via the storage object:

```
>>> sum = db.log.severity.sum()
>>> print db().select(sum).first()[sum]
6
```

You can also use `avg`, `min`, and `max` to the average, minimum, and maximum value respectively for the selected records. For example:

```
>>> max = db.log.severity.max()
>>> print db().select(max).first()[max]
3
```

`len` computes the length of field's value. It is generally used on string or text fields but depending on the back-end it may still work for other types too (boolean, integer, etc).

```
>>> for row in db(db.log.event.len() > 13).select():
...     print row.event
```

```
...
unauthorized login
```

Expressions can be combined to form more complex expressions. For example here we are computing the sum of the length of the event strings in the logs plus one:

```
>>> exp = (db.log.event.len() + 1).sum()
>>> db().select(exp).first()[exp]
43
```

### 6.25.5 Substrings

One can build an expression to refer to a substring. For example, we can group things whose name starts with the same three characters and select only one from each group:

```
db(db.thing).select(distinct = db.thing.name[:3])
```

### 6.25.6 Default values with `coalesce` and `coalesce_zero`

There are times when you need to pull a value from database but also need a default values if the value for a record is set to NULL. In SQL there is a function, `COALESCE`, for this. py4web has an equivalent `coalesce` method:

```
>>> db.define_table('sysuser', Field('username'), Field('fullname'))
<Table sysuser (id, username, fullname)>
>>> db.sysuser.insert(username='max', fullname='Max Power')
1
>>> db.sysuser.insert(username='tim', fullname=None)
2
>>> coa = db.sysuser.fullname.coalesce(db.sysuser.username)
>>> for row in db().select(coa):
...     print row[coa]
...
Max Power
tim
```

Other times you need to compute a mathematical expression but some fields have a value set to None while it should be zero. `coalesce_zero` comes to the rescue by defaulting None to zero in the query:

```
>>> db.define_table('sysuser', Field('username'), Field('points'))
<Table sysuser (id, username, points)>
>>> db.sysuser.insert(username='max', points=10)
1
>>> db.sysuser.insert(username='tim', points=None)
2
>>> exp = db.sysuser.points.coalesce_zero().sum()
>>> db().select(exp).first()[exp]
10
>>> type(exp)
<class 'pydal.objects.Expression'>
>>> print exp
SUM(COALESCE("sysuser"."points",'0'))
```

## 6.26 Generating raw sql

Sometimes you need to generate the SQL but not execute it. This is easy to do with py4web since every command that performs database IO has an equivalent command that does not, and simply returns the SQL that would have been executed. These commands have the same names and syntax as the functional ones, but they start with an underscore:

Here is `_insert`

```
>>> print db.person._insert(name='Alex')
INSERT INTO "person"("name") VALUES ('Alex');
```

Here is `_count`

```
>>> print db(db.person.name == 'Alex')._count()
SELECT COUNT(*) FROM "person" WHERE ("person"."name" = 'Alex');
```

Here is `_select`

```
>>> print db(db.person.name == 'Alex')._select()
SELECT "person"."id", "person"."name" FROM "person" WHERE ("person"."name" = 'Alex');
```

Here is `_delete`

```
>>> print db(db.person.name == 'Alex')._delete()
DELETE FROM "person" WHERE ("person"."name" = 'Alex');
```

And finally, here is `_update`

```
>>> print db(db.person.name == 'Alex')._update(name='Susan')
UPDATE "person" SET "name"='Susan' WHERE ("person"."name" = 'Alex');
```

> Moreover you can always use `db._lastsql` to return the most recent SQL code, whether it was executed manually using executesql or was SQL generated by the DAL.

## 6.27 Exporting and importing data

### 6.27.1 CSV (one Table at a time)

When a Rows object is converted to a string it is automatically serialized in CSV:

```
>>> rows = db(db.person.id == db.thing.owner_id).select()
>>> print rows
person.id,person.name,thing.id,thing.name,thing.owner_id
1,Alex,1,Boat,1
1,Alex,2,Chair,1
2,Bob,3,Shoes,2
```

You can serialize a single table in CSV and store it in a file "test.csv":

```
with open('test.csv', 'wb') as dumpfile:
    dumpfile.write(str(db(db.person).select()))
```

Notice that converting a `Rows` object into a string using Python 2 produces an utf8 encoded binary string. To obtain a different encoding you have to ask for it explicitly, for example with:

`unicode(str(db(db.person).select()), 'utf8').encode(...)`:pythonn

Or in Python 3:

```
with open('test.csv', 'w', encoding='utf-8', newline='') as dumpfile:
    dumpfile.write(str(db(db.person).select()))
```

This is equivalent to

```
rows = db(db.person).select()
with open('test.csv', 'wb') as dumpfile:
    rows.export_to_csv_file(dumpfile)
```

You can read the CSV file back with:

```
with open('test.csv', 'rb') as dumpfile:
    db.person.import_from_csv_file(dumpfile)
```

Again, when using Python 3, you can be explict about the encoding for the exporting file:

```
rows = db(db.person).select()
with open('test.csv', 'w', encoding='utf-8', newline='') as dumpfile:
    rows.export_to_csv_file(dumpfile)
```

and the importing one:

```
with open('test.csv', 'r', encoding='utf-8', newline='') as dumpfile:
    db.person.import_from_csv_file(dumpfile)
```

When importing, py4web looks for the field names in the CSV header. In this example, it finds two columns: "person.id" and "person.name". It ignores the "person." prefix, and it ignores the "id" fields. Then all records are appended and assigned new ids. Both of these operations can be performed via the appadmin web interface.

### 6.27.2 CSV (all tables at once)

In py4web, you can backup/restore an entire database with two commands:

To export:

```
with open('somefile.csv', 'wb') as dumpfile:
    db.export_to_csv_file(dumpfile)
```

To import:

```
with open('somefile.csv', 'rb') as dumpfile:
    db.import_from_csv_file(dumpfile)
```

Or in Python 3:

To export:

```
with open('somefile.csv', 'w', encoding='utf-8', newline='') as dumpfile:
    db.export_to_csv_file(dumpfile)
```

To import:

```
with open('somefile.csv', 'r', encoding='utf-8', newline='') as dumpfile:
    db.import_from_csv_file(dumpfile)
```

This mechanism can be used even if the importing database is of a different type than the exporting database.

The data is stored in "somefile.csv" as a CSV file where each table starts with one line that indicates the tablename, and another line with the fieldnames:

```
TABLE tablename
field1,field2,field3,...
```

Two tables are separated by `\r\n\r\n` (that is two empty lines). The file ends with the line

```
END
```

The file does not include uploaded files if these are not stored in the database. The upload files stored on filesystem must be dumped separately, a zip of the "uploads" folder may suffice in most cases.

When importing, the new records will be appended to the database if it is not empty. In general the new imported records will not have the same record id as the original (saved) records but py4web will restore references so they are not broken, even if the id values may change.

If a table contains a field called `uuid`, this field will be used to identify duplicates. Also, if an imported record has the same `uuid` as an existing record, the previous record will be updated.

### 6.27.3 CSV and remote database synchronization

Consider once again the following model:

```
db.define_table('person',
                Field('name'))

db.define_table('thing',
                Field('name'),
                Field('owner_id', 'reference person'))

# usage example
if db(db.person).isempty():
    nid = db.person.insert(name='Massimo')
    db.thing.insert(name='Chair', owner_id=nid)
```

Each record is identified by an identifier and referenced by that id. If you have two copies of the database used by distinct py4web installations, the id is unique only within each database and not across the databases. This is a problem when merging records from different databases.

In order to make records uniquely identifiable across databases, they must: - have a unique id (UUID), - have a last modification time to track the most recent among multiple copies, - reference the UUID instead of the id.

This can be achieved changing the above model into:

```
import uuid

db.define_table('person',
                Field('uuid', length=64),
                Field('modified_on', 'datetime', default=request.now,
update=request.now),
                Field('name'))
```

```
db.define_table('thing',
                Field('uuid', length=64),
                Field('modified_on', 'datetime', default=request.now,
update=request.now),
                Field('name'),
                Field('owner_id', length=64))

db.person.uuid.default = db.thing.uuid.default = lambda:str(uuid.uuid4())

db.thing.owner_id.requires = IS_IN_DB(db, 'person.uuid', '%(name)s')

# usage example
if db(db.person).isempty():
    nid = str(uuid.uuid4())
    db.person.insert(uuid=nid, name='Massimo')
    db.thing.insert(name='Chair', owner_id=nid)
```

Notice that in the above table definitions, the default value for the two `uuid` fields is set to a lambda function, which returns a UUID (converted to a string). The lambda function is called once for each record inserted, ensuring that each record gets a unique UUID, even if multiple records are inserted in a single transaction.

Create a controller action to export the database:

```
def export():
    s = StringIO.StringIO()
    db.export_to_csv_file(s)
    response.headers['Content-Type'] = 'text/csv'
    return s.getvalue()
```

Create a controller action to import a saved copy of the other database and sync records:

```
from yatl.helpers import FORM, INPUT

def import_and_sync():
    form = FORM(INPUT(_type='file', _name='data'), INPUT(_type='submit'))
    if form.process().accepted:
        db.import_from_csv_file(form.vars.data.file, unique=False)
        # for every table
        for tablename in db.tables:
            table = db[tablename]
            # for every uuid, delete all but the latest
            items = db(table).select(table.id, table.uuid,
                                     orderby=~table.modified_on,
                                     groupby=table.uuid)
            for item in items:
                db((table.uuid == item.uuid) & (table.id != item.id)).delete()
    return dict(form=form)
```

Optionally you should create an index manually to make the search by uuid faster.

Alternatively, you can use XML-RPC to export/import the file.

If the records reference uploaded files, you also need to export/import the content of the uploads folder. Notice that files therein are already labeled by UUIDs so you do not need to worry about naming conflicts and references.

### 6.27.4 HTML and XML (one Table at a time)

Rows objects also have an `xml` method (like helpers) that serializes it to XML/HTML:

```
>>> rows = db(db.person.id == db.thing.owner_id).select()
>>> print rows.xml()
```

```
<table>
<thead>
<tr><th>person.id</th><th>person.name</th><th>thing.id</th><th>thing.name</th><th>thing.owner_id</t
</thead>
<tbody>
<tr class="w2p_odd odd"><td>1</td><td>Alex</td><td>1</td><td>Boat</td><td>1</td></tr>
<tr class="w2p_even
even"><td>1</td><td>Alex</td><td>2</td><td>Chair</td><td>1</td></tr>
<tr class="w2p_odd odd"><td>2</td><td>Bob</td><td>3</td><td>Shoes</td><td>2</td></tr>
</tbody>
</table>
```

If you need to serialize the Rows in any other XML format with custom tags, you can easily do that using the universal `TAG` helper (described in Chapter 8 and the Python syntax `*<iterable>` allowed in function calls:

```
>>> rows = db(db.person).select()
>>> print TAG.result(*[TAG.row(*[TAG.field(r[f], _name=f) for f in db.person.fields])
 for r in rows])
```

```
<result>
<row><field name="id">1</field><field name="name">Alex</field></row>
<row><field name="id">2</field><field name="name">Bob</field></row>
<row><field name="id">3</field><field name="name">Carl</field></row>
</result>
```

### 6.27.5 Data representation

The `Rows.export_to_csv_file` method accepts a keyword argument named `represent`. When `True` it will use the columns `represent` function while exporting the data instead of the raw data.

The function also accepts a keyword argument named `colnames` that should contain a list of column names one wish to export. It defaults to all columns.

Both `export_to_csv_file` and `import_from_csv_file` accept keyword arguments that tell the csv parser the format to save/load the files: - `delimiter`: delimiter to separate values (default ',') - `quotechar`: character to use to quote string values (default to double quotes) - `quoting`: quote system (default `csv.QUOTE_MINIMAL`)

Here is some example usage:

```
import csv
rows = db(query).select()
with open('/tmp/test.txt', 'wb') as oufile:
    rows.export_to_csv_file(oufile,
                            delimiter='|',
                            quotechar='"',
                            quoting=csv.QUOTE_NONNUMERIC)
```

Which would render something similar to

```
"hello"|35|"this is the text description"|"2013-03-03"
```

For more information consult the official Python documentation

## 6.28 Caching selects

The select method also takes a `cache` argument, which defaults to None. For caching purposes, it should be set to a tuple where the first element is the cache model (`cache.ram`, `cache.disk`, etc.), and the second element is the expiration time in seconds.

In the following example, you see a controller that caches a select on the previously defined db.log table. The actual select fetches data from the back-end database no more frequently than once every 60 seconds and stores the result in memory. If the next call to this controller occurs in less than 60 seconds since the last database IO, it simply fetches the previous data from memory.

```
def cache_db_select():
    logs = db().select(db.log.ALL, cache=(cache.ram, 60))
    return dict(logs=logs)
```

The `select` method has an optional `cacheable` argument, normally set to False. When `cacheable=True` the resulting `Rows` is serializable but The `Rows` lack `update_record` and `delete_record` methods.

If you do not need these methods you can speed up selects a lot by setting the `cacheable` attribute:

```
rows = db(query).select(cacheable=True)
```

When the `cache` argument is set but `cacheable=False` (default) only the database results are cached, not the actual Rows object. When the `cache` argument is used in conjunction with `cacheable=True` the entire Rows object is cached and this results in much faster caching:

```
rows = db(query).select(cache=(cache.ram, 3600), cacheable=True)
```

## 6.29 Self-Reference and aliases

It is possible to define tables with fields that refer to themselves, here is an example:

```
db.define_table('person',
                Field('name'),
                Field('father_id', 'reference person'),
                Field('mother_id', 'reference person'))
```

Notice that the alternative notation of using a table object as field type will fail in this case, because it uses a table before it is defined:

```
db.define_table('person',
                Field('name'),
                Field('father_id', db.person),   # wrong!
                Field('mother_id', db['person']))   # wrong!
```

In general `db.tablename` and `'reference tablename'` are equivalent field types, but the latter is the only one allowed for self-references.

When a table has a self-reference and you have to do join, for example to select a person and its father,

you need an alias for the table. In SQL an alias is a temporary alternate name you can use to reference a table/column into a query (or other SQL statement).

With py4web you can make an alias for a table using the `with_alias` method. This works also for expressions, which means also for fields since `Field` is derived from `Expression`.

Here is an example:

```
>>> fid, mid = db.person.bulk_insert([dict(name='Massimo'), dict(name='Claudia')])
>>> db.person.insert(name='Marco', father_id=fid, mother_id=mid)
3
>>> Father = db.person.with_alias('father')
>>> Mother = db.person.with_alias('mother')
>>> type(Father)
<class 'pydal.objects.Table'>
>>> str(Father)
'person AS father'
>>> rows = db().select(db.person.name, Father.name, Mother.name,
...                    left=(Father.on(Father.id == db.person.father_id),
...                          Mother.on(Mother.id == db.person.mother_id)))
>>> for row in rows:
...     print row.person.name, row.father.name, row.mother.name
...
Massimo None None
Claudia None None
Marco Massimo Claudia
```

Notice that we have chosen to make a distinction between: - "father_id": the field name used in the table "person"; - "father": the alias we want to use for the table referenced by the above field; this is communicated to the database; - "Father": the variable used by py4web to refer to that alias.

The difference is subtle, and there is nothing wrong in using the same name for the three of them:

```
>>> db.define_table('person',
...                 Field('name'),
...                 Field('father', 'reference person'),
...                 Field('mother', 'reference person'))
<Table person (id, name, father, mother)>
>>> fid, mid = db.person.bulk_insert([dict(name='Massimo'), dict(name='Claudia')])
>>> db.person.insert(name='Marco', father=fid, mother=mid)
3
>>> father = db.person.with_alias('father')
>>> mother = db.person.with_alias('mother')
>>> rows = db().select(db.person.name, father.name, mother.name,
...                    left=(father.on(father.id==db.person.father),
...                          mother.on(mother.id==db.person.mother)))
>>> for row in rows:
...     print row.person.name, row.father.name, row.mother.name
...
Massimo None None
Claudia None None
Marco Massimo Claudia
```

But it is important to have the distinction clear in order to build correct queries.

## 6.30 Advanced features

### 6.30.1 Table inheritance

It is possible to create a table that contains all the fields from another table. It is sufficient to pass the other table in place of a field to `define_table`. For example

```
>>> db.define_table('person', Field('name'), Field('gender'))
<Table person (id, name, gender)>
>>> db.define_table('doctor', db.person, Field('specialization'))
<Table doctor (id, name, gender, specialization)>
```

It is also possible to define a dummy table that is not stored in a database in order to reuse it in multiple other places. For example:

```
signature = db.Table(db, 'signature',
                     Field('is_active', 'boolean', default=True),
                     Field('created_on', 'datetime', default=request.now),
                     Field('created_by', db.auth_user, default=auth.user_id),
                     Field('modified_on', 'datetime', update=request.now),
                     Field('modified_by', db.auth_user, update=auth.user_id))

db.define_table('payment', Field('amount', 'double'), signature)
```

This example assumes that standard py4web authentication is enabled.

Notice that if you use `Auth` py4web already creates one such table for you:

```
auth = Auth(db)
db.define_table('payment', Field('amount', 'double'), auth.signature)
```

When using table inheritance, if you want the inheriting table to inherit validators, be sure to define the validators of the parent table before defining the inheriting table.

### 6.30.2 `filter_in` and `filter_out`

It is possible to define a filter for each field to be called before a value is inserted into the database for that field and after a value is retrieved from the database.

Imagine for example that you want to store a serializable Python data structure in a field in the json format. Here is how it could be accomplished:

```
>>> import json
>>> db.define_table('anyobj',
...                 Field('name'),
...                 Field('data', 'text'))
<Table anyobj (id, name, data)>
>>> db.anyobj.data.filter_in = lambda obj: json.dumps(obj)
>>> db.anyobj.data.filter_out = lambda txt: json.loads(txt)
>>> myobj = ['hello', 'world', 1, {2: 3}]
>>> aid = db.anyobj.insert(name='myobjname', data=myobj)
>>> row = db.anyobj[aid]
>>> row.data
['hello', 'world', 1, {'2': 3}]
```

Another way to accomplish the same is by using a Field of type `SQLCustomType`, as discussed in next *Custom ``Field`` types <#Custom_Field_Types>`__* section.

### 6.30.3 callbacks on record insert, delete and update

PY4WEB provides a mechanism to register callbacks to be called before and/or after insert, update and delete of records.

Each table stores six lists of callbacks:

```
db.mytable._before_insert
db.mytable._after_insert
db.mytable._before_update
db.mytable._after_update
db.mytable._before_delete
db.mytable._after_delete
```

You can register a callback function by appending it to the corresponding list. The caveat is that depending on the functionality, the callback has different signature.

This is best explained via some examples.

```
>>> db.define_table('person', Field('name'))
<Table person (id, name)>
>>> def pprint(callback, *args):
...     print "%s%s" % (callback, args)
...
>>> db.person._before_insert.append(lambda f: pprint('before_insert', f))
>>> db.person._after_insert.append(lambda f, i: pprint('after_insert', f, i))
>>> db.person.insert(name='John')
before_insert(<OpRow {'name': 'John'}>,)
after_insert(<OpRow {'name': 'John'}>, 1L)
1L
>>> db.person._before_update.append(lambda s, f: pprint('before_update', s, f))
>>> db.person._after_update.append(lambda s, f: pprint('after_update', s, f))
>>> db(db.person.id == 1).update(name='Tim')
before_update(<Set ("person"."id" = 1)>, <OpRow {'name': 'Tim'}>)
after_update(<Set ("person"."id" = 1)>, <OpRow {'name': 'Tim'}>)
1
>>> db.person._before_delete.append(lambda s: pprint('before_delete', s))
>>> db.person._after_delete.append(lambda s: pprint('after_delete', s))
>>> db(db.person.id == 1).delete()
before_delete(<Set ("person"."id" = 1)>,)
after_delete(<Set ("person"."id" = 1)>,)
1
```

As you can see: - `f` gets passed the `OpRow` object with data for insert or update. - `i` gets passed the id of the newly inserted record. - `s` gets passed the `Set` object used for update or delete. `OpRow` is an helper object specialized in storing (field, value) pairs, you can think of it as a normal dictionary that you can use even with the syntax of attribute notation (that is `f.name` and `f['name']` are equivalent).

The return values of these callback should be `None` or `False`. If any of the `_before_*` callback returns a `True` value it will abort the actual insert/update/delete operation.

Some times a callback may need to perform an update in the same or a different table and one wants to avoid firing other callbacks, which could cause an infinite loop.

For this purpose there the `Set` objects have an `update_naive` method that works like `update` but ignores before and after callbacks.

#### Database cascades

Database schema can define relationships which trigger deletions of related records, known as cascading. The DAL is not informed when a record is deleted due to a cascade. So no *_delete callaback will ever be called as conseguence of a cascade-deletion.

### 6.30.4 Record versioning

It is possible to ask py4web to save every copy of a record when the record is individually modified. There are different ways to do it and it can be done for all tables at once using the syntax:

```
auth.enable_record_versioning(db)
```

this requires `Auth`. It can also be done for each individual table as discussed below.

Consider the following table:

```
db.define_table('stored_item',
                Field('name'),
                Field('quantity', 'integer'),
                Field('is_active', 'boolean',
                      writable=False, readable=False, default=True))
```

Notice the hidden boolean field called `is_active` and defaulting to True.

We can tell py4web to create a new table (in the same or a different database) and store all previous versions of each record in the table, when modified.

This is done in the following way:

```
db.stored_item._enable_record_versioning()
```

or in a more verbose syntax:

```
db.stored_item._enable_record_versioning(archive_db=db,
                                         archive_name='stored_item_archive',
                                         current_record='current_record',
                                         is_active='is_active')
```

The `archive_db=db` tells py4web to store the archive table in the same database as the `stored_item` table. The `archive_name` sets the name for the archive table. The archive table has the same fields as the original table `stored_item` except that unique fields are no longer unique (because it needs to store multiple versions) and has an extra field which name is specified by `current_record` and which is a reference to the current record in the `stored_item` table.

When records are deleted, they are not really deleted. A deleted record is copied in the `stored_item_archive` table (like when it is modified) and the `is_active` field is set to False. By enabling record versioning py4web sets a `common_filter` on this table that hides all records in table `stored_item` where the `is_active` field is set to False. The `is_active` parameter in the `_enable_record_versioning` method allows to specify the name of the field used by the `common_filter` to determine if the field was deleted or not.

`common_filters` will be discussed in next Common filters section.

### 6.30.5 Common fields and multi-tenancy

`db._common_fields` is a list of fields that should belong to all the tables. This list can also contain tables and it is understood as all fields from the table.

For example occasionally you find yourself in need to add a signature to all your tables but the `Auth` tables. In this case, after you `auth.define_tables()` but before defining any other table, insert:

```
db._common_fields.append(auth.signature)
```

One field is special: `request_tenant`, you can set a different name in `db._request_tenant`. This field does not exist but you can create it and add it to any of your tables (or all of them):

```
db._common_fields.append(Field('request_tenant',
                               default=request.env.http_host,
                               writable=False))
```

For every table with such a field, all records for all queries are always automatically filtered by:

```
db.table.request_tenant == db.table.request_tenant.default
```

and for every record inserted, this field is set to the default value. In the example above we have chosen:

```
default = request.env.http_host
```

this means we have chosen to ask our app to filter all tables in all queries with:

```
db.table.request_tenant == request.env.http_host
```

This simple trick allow us to turn any application into a multi-tenant application. Even though we run one instance of the application and we use one single database, when the application is accessed under two or more domains the visitors will see different data depending on the domain (in the example the domain name is retrieved from `request.env.http_host`).

You can turn off multi tenancy filters using `ignore_common_filters=True` at `Set` creation time:

```
db(query, ignore_common_filters=True)
```

### 6.30.6 Common filters

A common filter is a generalization of the above multi-tenancy idea. It provides an easy way to prevent repeating of the same query. Consider for example the following table:

```
db.define_table('blog_post',
                Field('subject'),
                Field('post_text', 'text'),
                Field('is_public', 'boolean'),
                common_filter = lambda query: db.blog_post.is_public == True)
```

Any select, delete or update in this table, will include only public blog posts. The attribute can also be modified at runtime:

```
db.blog_post._common_filter = lambda query: ...
```

It serves both as a way to avoid repeating the "db.blog_post.is_public==True" phrase in each blog post search, and also as a security enhancement, that prevents you from forgetting to disallow viewing of non-public posts.

In case you actually do want items left out by the common filter (for example, allowing the admin to see non-public posts), you can either remove the filter:

```
db.blog_post._common_filter = None
```

or ignore it:

```
db(query, ignore_common_filters=True)
```

Note that common_filters are ignored by the appadmin interface.

### 6.30.7 Custom `Field` types

Aside for using `filter_in` and `filter_out`, it is possible to define new/custom field types. For example, suppose that you want to define a custom type to store an IP address:

```
>>> def ip2int(sv):
...     "Convert an IPV4 to an integer."
...     sp = sv.split('.'); assert len(sp) == 4 # IPV4 only
...     iip = 0
...     for i in map(int, sp): iip = (iip<<8) + i
...     return iip
...
>>> def int2ip(iv):
...     "Convert an integer to an IPV4."
...     assert iv > 0
...     iv = (iv,); ov = []
...     for i in range(3):
...         iv = divmod(iv[0], 256)
...         ov.insert(0, iv[1])
...     ov.insert(0, iv[0])
...     return '.'.join(map(str, ov))
...
>>> from gluon.dal import SQLCustomType
>>> ipv4 = SQLCustomType(type='string', native='integer',
...                      encoder=lambda x : str(ip2int(x)), decoder=int2ip)
>>> db.define_table('website',
...                 Field('name'),
...                 Field('ipaddr', type=ipv4))
<Table website (id, name, ipaddr)>
>>> db.website.insert(name='wikipedia', ipaddr='91.198.174.192')
1
>>> db.website.insert(name='google', ipaddr='172.217.11.174')
2
>>> db.website.insert(name='youtube', ipaddr='74.125.65.91')
3
>>> db.website.insert(name='github', ipaddr='207.97.227.239')
4
>>> rows = db(db.website.ipaddr > '100.0.0.0').select(orderby=~db.website.ipaddr)
>>> for row in rows:
...     print row.name, row.ipaddr
...
github 207.97.227.239
google 172.217.11.174
```

`SQLCustomType` is a field type factory. Its `type` argument must be one of the standard py4web types. It tells py4web how to treat the field values at the py4web level. `native` is the type of the field as far as the database is concerned. Allowed names depend on the database engine. `encoder` is an optional transformation function applied when the data is stored and `decoder` is the optional reverse transformation function.

This feature is marked as experimental. In practice it has been in py4web for a long time and it works but it can make the code not portable, for example when the native type is database specific.

It does not work on Google App Engine NoSQL.

### 6.30.8 Using DAL without define tables

The DAL can be used from any Python program simply by doing this:

```
from gluon import DAL
db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases')
```

i.e. import the DAL, connect and specify the folder which contains the .table files (the app/databases folder).

To access the data and its attributes we still have to define all the tables we are going to access with `db.define_table`.

If we just need access to the data but not to the py4web table attributes, we get away without re-defining the tables but simply asking py4web to read the necessary info from the metadata in the .table files:

```
from gluon import DAL
db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases', auto_import=True)
```

This allows us to access any db.table without need to re-define it.

### 6.30.9 PostGIS, SpatiaLite, and MS Geo (experimental)

The DAL supports geographical APIs using PostGIS (for PostgreSQL), SpatiaLite (for SQLite), and MSSQL and Spatial Extensions. This is a feature that was sponsored by the Sahana project and implemented by Denes Lengyel.

DAL provides geometry and geography fields types and the following functions:

```
st_asgeojson (PostGIS only)
st_astext
st_contains
st_distance
st_equals
st_intersects
st_overlaps
st_simplify (PostGIS only)
st_touches
st_within
st_x
st_y
```

Here are some examples:

```
>>> from gluon.dal import DAL, Field, geoPoint, geoLine, geoPolygon
>>> db = DAL("mssql://user:pass@host/db")
>>> sp = db.define_table('spatial', Field('loc', 'geometry()'))
```

Below we insert a point, a line, and a polygon:

```
>>> sp.insert(loc=geoPoint(1, 1))
1
>>> sp.insert(loc=geoLine((100, 100), (20, 180), (180, 180)))
2
>>> sp.insert(loc=geoPolygon((0, 0), (150, 0), (150, 150), (0, 150), (0, 0)))
3
```

Notice that

```
rows = db(sp).select()
```

Always returns the geometry data serialized as text. You can also do the same more explicitly using `st_astext()`:

```
>>> print db(sp).select(sp.id, sp.loc.st_astext())
spatial.id,spatial.loc.STAsText()
1,"POINT (1 2)"
2,"LINESTRING (100 100, 20 180, 180 180)"
3,"POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))"
```

You can ask for the native representation by using `st_asgeojson()` (in PostGIS only):

```
>>> print db(sp).select(sp.id, sp.loc.st_asgeojson().with_alias('loc'))
spatial.id,loc
1,[1, 2]
2,[[100, 100], [20 180], [180, 180]]
3,[[[0, 0], [150, 0], [150, 150], [0, 150], [0, 0]]]
```

(notice an array is a point, an array of arrays is a line, and an array of array of arrays is a polygon).

Here are example of how to use geographical functions:

```
>>> query = sp.loc.st_intersects(geoLine((20, 120), (60, 160)))
>>> query = sp.loc.st_overlaps(geoPolygon((1, 1), (11, 1), (11, 11), (11, 1), (1,
1)))
>>> query = sp.loc.st_contains(geoPoint(1, 1))
>>> print db(query).select(sp.id, sp.loc)
spatial.id,spatial.loc
3,"POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))"
```

Computed distances can also be retrieved as floating point numbers:

```
>>> dist = sp.loc.st_distance(geoPoint(-1,2)).with_alias('dist')
>>> print db(sp).select(sp.id, dist)
spatial.id,dist
1,2.0
2,140.714249456
3,1.0
```

### 6.30.10 Copy data from one db into another

Consider the situation in which you have been using the following database:

```
db = DAL('sqlite://storage.sqlite')
```

and you wish to move to another database using a different connection string:

```
db = DAL('postgres://username:password@localhost/mydb')
```

Before you switch, you want to move the data and rebuild all the metadata for the new database. We assume the new database to exist but we also assume it is empty.

PY4WEB provides a script that does this work for you:

```
cd py4web
```

```
python scripts/cpdb.py \\
   -f applications/app/databases \\
   -y 'sqlite://storage.sqlite' \\
   -Y 'postgres://username:password@localhost/mydb' \\
   -d ../gluon
```

After running the script you can simply switch the connection string in the model and everything should work out of the box. The new data should be there.

This script provides various command line options that allows you to move data from one application to another, move all tables or only some tables, clear the data in the tables. For more info try:

```
python scripts/cpdb.py -h
```

### 6.30.11 Note on new DAL and adapters

The source code of the Database Abstraction Layer was completely rewritten in 2010. While it stays backward compatible, the rewrite made it more modular and easier to extend. Here we explain the main logic.

The file "gluon/dal.py" defines, among other, the following classes.

```
ConnectionPool
BaseAdapter extends ConnectionPool
Row
DAL
Reference
Table
Expression
Field
Query
Set
Rows
```

Their use has been explained in the previous sections, except for `BaseAdapter`. When the methods of a `Table` or `Set` object need to communicate with the database they delegate to methods of the adapter the task to generate the SQL and or the function call.

For example:

```
db.mytable.insert(myfield='myvalue')
```

calls

```
Table.insert(myfield='myvalue')
```

which delegates the adapter by returning:

```
db._adapter.insert(db.mytable, db.mytable._listify(dict(myfield='myvalue')))
```

Here `db.mytable._listify` converts the dict of arguments into a list of `(field, value)` and calls the `insert` method of the `adapter`. `db._adapter` does more or less the following:

```
query = db._adapter._insert(db.mytable, list_of_fields)
db._adapter.execute(query)
```

where the first line builds the query and the second executes it.

`BaseAdapter` defines the interface for all adapters.

"gluon/dal.py" at the moment of writing this book, contains the following adapters:

```
SQLiteAdapter extends BaseAdapter
JDBCSQLiteAdapter extends SQLiteAdapter
MySQLAdapter extends BaseAdapter
PostgreSQLAdapter extends BaseAdapter
JDBCPostgreSQLAdapter extends PostgreSQLAdapter
OracleAdapter extends BaseAdapter
MSSQLAdapter extends BaseAdapter
MSSQL2Adapter extends MSSQLAdapter
MSSQL3Adapter extends MSSQLAdapter
MSSQL4Adapter extends MSSQLAdapter
FireBirdAdapter extends BaseAdapter
FireBirdEmbeddedAdapter extends FireBirdAdapter
InformixAdapter extends BaseAdapter
DB2Adapter extends BaseAdapter
IngresAdapter extends BaseAdapter
IngresUnicodeAdapter extends IngresAdapter
GoogleSQLAdapter extends MySQLAdapter
NoSQLAdapter extends BaseAdapter
GoogleDatastoreAdapter extends NoSQLAdapter
CubridAdapter extends MySQLAdapter (experimental)
TeradataAdapter extends DB2Adapter (experimental)
SAPDBAdapter extends BaseAdapter (experimental)
CouchDBAdapter extends NoSQLAdapter (experimental)
IMAPAdapter extends NoSQLAdapter (experimental)
MongoDBAdapter extends NoSQLAdapter (experimental)
VerticaAdapter extends MSSQLAdapter (experimental)
SybaseAdapter extends MSSQLAdapter (experimental)
```

which override the behavior of the `BaseAdapter`.

Each adapter has more or less this structure:

```
class MySQLAdapter(BaseAdapter):

    # specify a driver to use
    driver = globals().get('pymysql', None)

    # map py4web types into database types
    types = {
        'boolean': 'CHAR(1)',
        'string': 'VARCHAR(%(length)s)',
        'text': 'LONGTEXT',
        ...
        }

    # connect to the database using driver
    def __init__(self, db, uri, pool_size=0, folder=None, db_codec ='UTF-8',
                 credential_decoder=lambda x:x, driver_args={},
                 adapter_args={}):
        # parse uri string and store parameters in driver_args
        ...
        # define a connection function
        def connect(driver_args=driver_args):
            return self.driver.connect(**driver_args)
        # place it in the pool
        self.pool_connection(connect)
        # set optional parameters (after connection)
        self.execute('SET FOREIGN_KEY_CHECKS=1;')
        self.execute("SET sql_mode='NO_BACKSLASH_ESCAPES';")
```

```
    # override BaseAdapter methods as needed
    def lastrowid(self, table):
        self.execute('select last_insert_id();')
        return int(self.cursor.fetchone()[0])
```

Looking at the various adapters as example should be easy to write new ones.

When db instance is created:

```
db = DAL('mysql://...')
```

the prefix in the uri string defines the adapter. The mapping is defined in the following dictionary also in "gluon/dal.py":

```
ADAPTERS = {
    'sqlite': SQLiteAdapter,
    'spatialite': SpatiaLiteAdapter,
    'sqlite:memory': SQLiteAdapter,
    'spatialite:memory': SpatiaLiteAdapter,
    'mysql': MySQLAdapter,
    'postgres': PostgreSQLAdapter,
    'postgres:psycopg2': PostgreSQLAdapter,
    'postgres2:psycopg2': NewPostgreSQLAdapter,
    'oracle': OracleAdapter,
    'mssql': MSSQLAdapter,
    'mssql2': MSSQL2Adapter,
    'mssql3': MSSQL3Adapter,
    'mssql4' : MSSQL4Adapter,
    'vertica': VerticaAdapter,
    'sybase': SybaseAdapter,
    'db2': DB2Adapter,
    'teradata': TeradataAdapter,
    'informix': InformixAdapter,
    'informix-se': InformixSEAdapter,
    'firebird': FireBirdAdapter,
    'firebird_embedded': FireBirdAdapter,
    'ingres': IngresAdapter,
    'ingresu': IngresUnicodeAdapter,
    'sapdb': SAPDBAdapter,
    'cubrid': CubridAdapter,
    'jdbc:sqlite': JDBCSQLiteAdapter,
    'jdbc:sqlite:memory': JDBCSQLiteAdapter,
    'jdbc:postgres': JDBCPostgreSQLAdapter,
    'gae': GoogleDatastoreAdapter, # discouraged, for backward compatibility
    'google:datastore': GoogleDatastoreAdapter,
    'google:datastore+ndb': GoogleDatastoreAdapter,
    'google:sql': GoogleSQLAdapter,
    'couchdb': CouchDBAdapter,
    'mongodb': MongoDBAdapter,
    'imap': IMAPAdapter
}
```

the uri string is then parsed in more detail by the adapter itself.

For any adapter you can replace the driver with a different one:

```
import MySQLdb as mysqldb
from gluon.dal import MySQLAdapter
```

```
MySQLAdapter.driver = mysqldb
```

i.e. `mysqldb` has to be *that module* with a .connect() method. You can specify optional driver arguments and adapter arguments:

```
db =DAL(..., driver_args={}, adapter_args={})
```

## 6.31 Gotchas

### 6.31.1 SQLite

SQLite does not support dropping and altering columns. That means that py4web migrations will work up to a point. If you delete a field from a table, the column will remain in the database but will be invisible to py4web. If you decide to reinstate the column, py4web will try re-create it and fail. In this case you must set `fake_migrate=True` so that metadata is rebuilt without attempting to add the column again. Also, for the same reason, **SQLite** is not aware of any change of column type. If you insert a number in a string field, it will be stored as string. If you later change the model and replace the type "string" with type "integer", SQLite will continue to keep the number as a string and this may cause problem when you try to extract the data.

SQLite doesn't have a boolean type. py4web internally maps booleans to a 1 character string, with 'T' and 'F' representing True and False. The DAL handles this completely; the abstraction of a true boolean value works well. But if you are updating the SQLite table with SQL directly, be aware of the py4web implementation, and avoid using 0 and 1 values.

### 6.31.2 MySQL

MySQL does not support multiple ALTER TABLE within a single transaction. This means that any migration process is broken into multiple commits. If something happens that causes a failure it is possible to break a migration (the py4web metadata are no longer in sync with the actual table structure in the database). This is unfortunate but it can be prevented (migrate one table at the time) or it can be fixed a posteriori (revert the py4web model to what corresponds to the table structure in database, set `fake_migrate=True` and after the metadata has been rebuilt, set `fake_migrate=False` and migrate the table again).

### 6.31.3 Google SQL

Google SQL has the same problems as MySQL and more. In particular table metadata itself must be stored in the database in a table that is not migrated by py4web. This is because Google App Engine has a read-only file system. PY4WEB migrations in Google SQL combined with the MySQL issue described above can result in metadata corruption. Again, this can be prevented (by migrating the table at once and then setting migrate=False so that the metadata table is not accessed any more) or it can fixed a posteriori (by accessing the database using the Google dashboard and deleting any corrupted entry from the table called `py4web_filesystem`.

### 6.31.4 MSSQL (Microsoft SQL Server)

MSSQL < 2012 does not support the SQL OFFSET keyword. Therefore the database cannot do pagination. When doing a `limitby=(a, b)` py4web will fetch the first `a + b` rows and discard the first `a`. This may result in a considerable overhead when compared with other database engines. If you're using MSSQL >= 2005, the recommended prefix to use is `mssql3://` which provides a method to avoid the issue of fetching the entire non-paginated resultset. If you're on MSSQL >= 2012, use `mssql4://` that uses the `OFFSET ... ROWS ... FETCH NEXT ... ROWS ONLY` construct to support natively pagination

without performance hits like other backends. The `mssql://` uri also enforces (for historical reasons) the use of `text` columns, that are superseeded in more recent versions (from 2005 onwards) by `varchar(max)`. `mssql3://` and `mssql4://` should be used if you don't want to face some limitations of the - officially deprecated - `text` columns.

MSSQL has problems with circular references in tables that have ONDELETE CASCADE. This is an MSSQL bug and you work around it by setting the ondelete attribute for all reference fields to "NO ACTION". You can also do it once and for all before you define tables:

```
db = DAL('mssql://....')
for key in db._adapter.types:
    if ' ON DELETE %(on_delete_action)s' in db._adapter.types[key]:
        db._adapter.types[key] =
db._adapter.types[key].replace('%(on_delete_action)s', 'NO ACTION')
```

MSSQL also has problems with arguments passed to the DISTINCT keyword and therefore while this works,

```
db(query).select(distinct=True)
```

this does not

```
db(query).select(distinct=db.mytable.myfield)
```

### 6.31.5 Oracle

Oracle also does not support pagination. It does not support neither the OFFSET nor the LIMIT keywords. PY4WEB achieves pagination by translating a `db(...).select(limitby=(a, b))` into a complex three-way nested select (as suggested by official Oracle documentation). This works for simple select but may break for complex selects involving aliased fields and or joins.

### 6.31.6 Google NoSQL (Datastore)

Google NoSQL (Datastore) does not allow joins, left joins, aggregates, expression, OR involving more than one table, the 'like' operator searches in "text" fields.

Transactions are limited and not provided automatically by py4web (you need to use the Google API `run_in_transaction` which you can look up in the Google App Engine documentation online).

Google also limits the number of records you can retrieve in each one query (1000 at the time of writing). On the Google datastore record IDs are integer but they are not sequential. While on SQL the "list:string" type is mapped into a "text" type, on the Google Datastore it is mapped into a `ListStringProperty`. Similarly "list:integer" and "list:reference" are mapped into `ListProperty`. This makes searches for content inside these fields types more efficient on Google NoSQL than on SQL databases.

Chapter 7

# The RESTAPI

Since version 19.5.10 PyDAL includes a restful API called RestAPI. It is inspired by GraphQL but it's not quite the same because it is less powerful but, in the spirit of web2py, more practical and easier to use. Like GraphSQL RestAPI allows a client to query for information using the GET method and allows to specify some details about the format of the response (which references to follow, and how to denormalize the data). Unlike GraphSQL it allows the server to specify a policy and restrict which queries are allowed and which one are not. They can be evaluated dynamically per request based on the user and the state of the server. As the name implied RestAPI allows all stardard methods GET, POST, PUT, and DELETE. Each of them can be enabled or disabled based on the policy, for individual tables and individual fields.

In the examples below we assume an app called "superheroes" and the following model:

```
db.define_table(
    'person',
    Field('name'),
    Field('job'))

db.define_table(
    'superhero',
    Field('name'),
    Field('real_identity', 'reference person'))

db.define_table(
    'superpower',
    Field('description'))

db.define_table(
    'tag',
    Field('superhero', 'reference superhero'),
    Field('superpower', 'reference superpower'),
    Field('strength', 'integer'))
```

We also assume the following controller `rest.py`:

```
from pydal.dbapi import RestAPI, Policy

policy = Policy()
policy.set('superhero', 'GET', authorize=True, allowed_patterns=['*'])
policy.set('*', 'GET', authorize=True, allowed_patterns=['*'])
policy.set('*', 'PUT', authorize=False)
policy.set('*', 'POST', authorize=False)
policy.set('*', 'DELETE', authorize=False)
```

```
@action('api/<tablename>/', method = ['GET', 'POST'])
@action('api/<tablename>/<rec_id>', method = ['GET', 'PUT', 'DELETE'])
def api(tablename, rec_id=None):
    return RestAPI(db, policy)(request.method,
                               tablename,
                               rec_id,
                               request.GET,
                               request.POST
                               )
```

The policy is per table (or * for all tables and per method. authorize can be True (allow), False (deny) or a function with the signature (method, tablename, record_id, get_vars, post_vars) which returns True/-False. For the GET policy one can specify a list of allowed query patterns (* for all). A query pattern will be matched against the keys in the query string.

The above action is exposed as:

```
/superheroes/rest/api/{tablename}
```

In our example policy we disabled all methods but GET.

## 7.1 RestAPI GET

The general query has the form `{something}.eq=value` where `eq=` stands for "equal", `gt=` stands for "greater than", etc. The expression can be prepended by `not.`.

`{something}` can be the name of a field in the table been queried as in:

**All superheroes called "Superman"**

```
/superheroes/rest/api/superhero?name.eq=Superman
```

It can be a the name of a field of a table referred by the table been queried as in:

**All superheroes with real identity "Clark Kent"**

```
/superheroes/rest/api/superhero?real_identity.name.eq=Clark Kent
```

It can be the name of a field of a table that refers to the table neen queried as in:

**All superheroes with any tag superpower with strength > 90**

```
/superheroes/rest/api/superhero?superhero.tag.strength.gt=90
```

(here tag is the name of the link table, the preceding `superhero` is the name of the field that refers to the selected table and `strength` is the name of the field used to filter)

It can also be a field of the table referenced by a many-to-many linked table as in:

**All superheroes with the flight power**

```
/superheroes/rest/api/superhero?superhero.tag.superpower.description.eq=Flight
```

The key to understand the syntax above is to break it as follows:

```
superhero?superhero.tag.superpower.description.eq=Flight
```

and read it as:

select records of table **superhero** referred by field **superhero** of table **tag** when the **superpower** field of said table points to a record with **description eq**ual to "Flight".

The query allows additional modifiers for example

```
@offset=10
@limit=10
@order=name
@model=true
@lookup=real_identity
```

The first 3 are obvious. @model returns a JSON description of database model. Lookup denormalizes the linked field.

Here are some practical examples:

URL:

```
/superheroes/rest/api/superhero
```

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
            "id": 1
        },
        {
            "real_identity": 2,
            "name": "Spiderman",
            "id": 2
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.132635",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@model=true
```

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
```

```
            "real_identity": 1,
            "name": "Superman",
            "id": 1
        },
        {
            "real_identity": 2,
            "name": "Spiderman",
            "id": 2
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.098292",
    "model": [
        {
            "regex": "[1-9]\\d*",
            "name": "id",
            "default": null,
            "required": false,
            "label": "Id",
            "post_writable": true,
            "referenced_by": [],
            "unique": false,
            "type": "id",
            "options": null,
            "put_writable": true
        },
        {
            "regex": null,
            "name": "name",
            "default": null,
            "required": false,
            "label": "Name",
            "post_writable": true,
            "unique": false,
            "type": "string",
            "options": null,
            "put_writable": true
        },
        {
            "regex": null,
            "name": "real_identity",
            "default": null,
            "required": false,
            "label": "Real Identity",
            "post_writable": true,
            "references": "person",
            "unique": false,
            "type": "reference",
            "options": null,
            "put_writable": true
        }
    ],
    "api_version": "0.1"
}
```

URL:

```
/superheroes/rest/api/superhero?@lookup=real_identity
```

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": {
                "name": "Clark Kent",
                "job": "Journalist",
                "id": 1
            },
            "name": "Superman",
            "id": 1
        },
        {
            "real_identity": {
                "name": "Peter Park",
                "job": "Photographer",
                "id": 2
            },
            "name": "Spiderman",
            "id": 2
        },
        {
            "real_identity": {
                "name": "Bruce Wayne",
                "job": "CEO",
                "id": 3
            },
            "name": "Batman",
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.178974",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@lookup=identity:real_identity
```

(denormalize the real_identity and rename it identity)

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
```

```
            "id": 1,
            "identity": {
                "name": "Clark Kent",
                "job": "Journalist",
                "id": 1
            }
        },
        {
            "real_identity": 2,
            "name": "Spiderman",
            "id": 2,
            "identity": {
                "name": "Peter Park",
                "job": "Photographer",
                "id": 2
            }
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "id": 3,
            "identity": {
                "name": "Bruce Wayne",
                "job": "CEO",
                "id": 3
            }
        }
    ],
    "timestamp": "2019-05-19T05:38:00.123218",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@lookup=identity!:real_identity[name,job]
```

(denormalize the real_identity [but only fields name and job], collapse the with the identity prefix)

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "name": "Superman",
            "identity_job": "Journalist",
            "identity_name": "Clark Kent",
            "id": 1
        },
        {
            "name": "Spiderman",
            "identity_job": "Photographer",
            "identity_name": "Peter Park",
            "id": 2
        },
        {
            "name": "Batman",
```

```
            "identity_job": "CEO",
            "identity_name": "Bruce Wayne",
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.192180",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@lookup=superhero.tag
```

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
            "superhero.tag": [
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 1,
                    "superpower": 1
                },
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 2,
                    "superpower": 2
                },
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 3,
                    "superpower": 3
                },
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 4,
                    "superpower": 4
                }
            ],
            "id": 1
        },
        {
            "real_identity": 2,
            "name": "Spiderman",
            "superhero.tag": [
                {
                    "strength": 50,
                    "superhero": 2,
                    "id": 5,
```

```
                    "superpower": 2
                },
                {
                    "strength": 75,
                    "superhero": 2,
                    "id": 6,
                    "superpower": 3
                },
                {
                    "strength": 10,
                    "superhero": 2,
                    "id": 7,
                    "superpower": 4
                }
            ],
            "id": 2
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "superhero.tag": [
                {
                    "strength": 80,
                    "superhero": 3,
                    "id": 8,
                    "superpower": 2
                },
                {
                    "strength": 20,
                    "superhero": 3,
                    "id": 9,
                    "superpower": 3
                },
                {
                    "strength": 70,
                    "superhero": 3,
                    "id": 10,
                    "superpower": 4
                }
            ],
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.201988",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@lookup=superhero.tag.superpower
```

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
```

```
            "real_identity": 1,
            "name": "Superman",
            "superhero.tag.superpower": [
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 1,
                    "superpower": {
                        "id": 1,
                        "description": "Flight"
                    }
                },
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 2,
                    "superpower": {
                        "id": 2,
                        "description": "Strength"
                    }
                },
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 3,
                    "superpower": {
                        "id": 3,
                        "description": "Speed"
                    }
                },
                {
                    "strength": 100,
                    "superhero": 1,
                    "id": 4,
                    "superpower": {
                        "id": 4,
                        "description": "Durability"
                    }
                }
            ],
            "id": 1
        },
        {
            "real_identity": 2,
            "name": "Spiderman",
            "superhero.tag.superpower": [
                {
                    "strength": 50,
                    "superhero": 2,
                    "id": 5,
                    "superpower": {
                        "id": 2,
                        "description": "Strength"
                    }
                },
                {
                    "strength": 75,
                    "superhero": 2,
                    "id": 6,
```

```
                    "superpower": {
                        "id": 3,
                        "description": "Speed"
                    }
                },
                {
                    "strength": 10,
                    "superhero": 2,
                    "id": 7,
                    "superpower": {
                        "id": 4,
                        "description": "Durability"
                    }
                }
            ],
            "id": 2
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "superhero.tag.superpower": [
                {
                    "strength": 80,
                    "superhero": 3,
                    "id": 8,
                    "superpower": {
                        "id": 2,
                        "description": "Strength"
                    }
                },
                {
                    "strength": 20,
                    "superhero": 3,
                    "id": 9,
                    "superpower": {
                        "id": 3,
                        "description": "Speed"
                    }
                },
                {
                    "strength": 70,
                    "superhero": 3,
                    "id": 10,
                    "superpower": {
                        "id": 4,
                        "description": "Durability"
                    }
                }
            ],
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.322494",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@lookup=powers:superhero.tag[strength].superpower[description]
```

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
            "powers": [
                {
                    "strength": 100,
                    "superpower": {
                        "description": "Flight"
                    }
                },
                {
                    "strength": 100,
                    "superpower": {
                        "description": "Strength"
                    }
                },
                {
                    "strength": 100,
                    "superpower": {
                        "description": "Speed"
                    }
                },
                {
                    "strength": 100,
                    "superpower": {
                        "description": "Durability"
                    }
                }
            ],
            "id": 1
        },
        {
            "real_identity": 2,
            "name": "Spiderman",
            "powers": [
                {
                    "strength": 50,
                    "superpower": {
                        "description": "Strength"
                    }
                },
                {
                    "strength": 75,
                    "superpower": {
                        "description": "Speed"
                    }
                },
                {
                    "strength": 10,
                    "superpower": {
                        "description": "Durability"
                    }
```

```
                }
            ],
            "id": 2
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "powers": [
                {
                    "strength": 80,
                    "superpower": {
                        "description": "Strength"
                    }
                },
                {
                    "strength": 20,
                    "superpower": {
                        "description": "Speed"
                    }
                },
                {
                    "strength": 70,
                    "superpower": {
                        "description": "Durability"
                    }
                }
            ],
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.309903",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@lookup=powers!:superhero.tag[strength].superpower[description]
```

OUTPUT:


```
{
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
            "powers": [
                {
                    "strength": 100,
                    "description": "Flight"
                },
                {
                    "strength": 100,
                    "description": "Strength"
                },
                {
                    "strength": 100,
```

```
                    "description": "Speed"
                },
                {
                    "strength": 100,
                    "description": "Durability"
                }
            ],
            "id": 1
        },
        {
            "real_identity": 2,
            "name": "Spiderman",
            "powers": [
                {
                    "strength": 50,
                    "description": "Strength"
                },
                {
                    "strength": 75,
                    "description": "Speed"
                },
                {
                    "strength": 10,
                    "description": "Durability"
                }
            ],
            "id": 2
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "powers": [
                {
                    "strength": 80,
                    "description": "Strength"
                },
                {
                    "strength": 20,
                    "description": "Speed"
                },
                {
                    "strength": 70,
                    "description": "Durability"
                }
            ],
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.355181",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?@lookup=powers!:superhero.tag[strength].superpower[description],ide
```

OUTPUT:


```
{
```

```
    "count": 3,
    "status": "success",
    "code": 200,
    "items": [
        {
            "name": "Superman",
            "identity_name": "Clark Kent",
            "powers": [
                {
                    "strength": 100,
                    "description": "Flight"
                },
                {
                    "strength": 100,
                    "description": "Strength"
                },
                {
                    "strength": 100,
                    "description": "Speed"
                },
                {
                    "strength": 100,
                    "description": "Durability"
                }
            ],
            "id": 1
        },
        {
            "name": "Spiderman",
            "identity_name": "Peter Park",
            "powers": [
                {
                    "strength": 50,
                    "description": "Strength"
                },
                {
                    "strength": 75,
                    "description": "Speed"
                },
                {
                    "strength": 10,
                    "description": "Durability"
                }
            ],
            "id": 2
        },
        {
            "name": "Batman",
            "identity_name": "Bruce Wayne",
            "powers": [
                {
                    "strength": 80,
                    "description": "Strength"
                },
                {
                    "strength": 20,
                    "description": "Speed"
                },
                {
```

```
                    "strength": 70,
                    "description": "Durability"
                }
            ],
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.396583",
    "api_version": "0.1"
}
```

URL:

```
/superheroes/rest/api/superhero?name.eq=Superman
```

OUTPUT:

```
{
    "count": 1,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
            "id": 1
        }
    ],
    "timestamp": "2019-05-19T05:38:00.405515",
    "api_version": "0.1"
}
```

URL:

```
/superheroes/rest/api/superhero?real_identity.name.eq=Clark Kent
```

OUTPUT:

```
{
    "count": 1,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
            "id": 1
        }
    ],
    "timestamp": "2019-05-19T05:38:00.366288",
    "api_version": "0.1"
}
```

URL:

```
/superheroes/rest/api/superhero?not.real_identity.name.eq=Clark Kent
```

OUTPUT:

```
{
    "count": 2,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 2,
            "name": "Spiderman",
            "id": 2
        },
        {
            "real_identity": 3,
            "name": "Batman",
            "id": 3
        }
    ],
    "timestamp": "2019-05-19T05:38:00.451907",
    "api_version": "0.1"
}
```


URL:

```
/superheroes/rest/api/superhero?superhero.tag.superpower.description=Flight
```

OUTPUT:


```
{
    "count": 1,
    "status": "success",
    "code": 200,
    "items": [
        {
            "real_identity": 1,
            "name": "Superman",
            "id": 1
        }
    ],
    "timestamp": "2019-05-19T05:38:00.453020",
    "api_version": "0.1"
}
```


Notice all RestAPI response have the fields

```
{
    "api_version": ...
    "timestamp": ...
    "status": ...
    "code": ...
}
```

and some optional fields:

```
{
    "count": ... (total matching, not total returned, for GET)
    "items": ... (in response to a GET)
    "errors": ... (usually validation error0
    "models": ... (usually if status != success)
    "message": ... (is if error)
```

```
}
```

The exact specs are subject to change since this is a new feature.

Chapter 8

# YATL Template Language

py4web uses Python for its models, controllers, and views, although it uses a slightly modified Python syntax in the views to allow more readable code without imposing any restrictions on proper Python usage.

py4web uses `[[ ... ]]` to escape Python code embedded in HTML. The advantage of using square brackets instead of angle brackets is that it's transparent to all common HTML editors. This allows the developer to use those editors to create py4web views.

Since the developer is embedding Python code into HTML, the document should be indented according to HTML rules, and not Python rules. Therefore, we allow unindented Python inside the `[[ ... ]]` tags. Since Python normally uses indentation to delimit blocks of code, we need a different way to delimit them; this is why the py4web template language makes use of the Python keyword `pass`.

> A code block starts with a line ending with a colon and ends with a line beginning with `pass`. The keyword `pass` is not necessary when the end of the block is obvious from the context.

Here is an example:

```
[[
if i == 0:
response.write('i is 0')
else:
response.write('i is not 0')
pass
]]
```

Note that `pass` is a Python keyword, not a py4web keyword. Some Python editors, such as Emacs, use the keyword `pass` to signify the division of blocks and use it to re-indent code automatically.

The py4web template language does exactly the same. When it finds something like:

```
<html><body>
[[for x in range(10):]][[=x]]hello<br />[[pass]]
</body></html>
```

it translates it into a program:

```
response.write("""<html><body>""", escape=False)
for x in range(10):
    response.write(x)
    response.write("""hello<br />""", escape=False)
response.write("""</body></html>""", escape=False)
```

`response.write` writes to the `response.body`.

When there is an error in a py4web view, the error report shows the generated view code, not the actual view as written by the developer. This helps the developer debug the code by highlighting the actual code that is executed (which is something that can be debugged with an HTML editor or the DOM inspector of the browser).

Also note that:

```
[[=x]]
```

generates

```
response.write(x)
```

Variables injected into the HTML in this way are escaped by default. The escaping is ignored if `x` is an `XML` object, even if escape is set to `True`.

Here is an example that introduces the `H1` helper:

```
[[=H1(i)]]
```

which is translated to:

```
response.write(H1(i))
```

upon evaluation, the `H1` object and its components are recursively serialized, escaped and written to the response body. The tags generated by `H1` and inner HTML are not escaped. This mechanism guarantees that all text — and only text — displayed on the web page is always escaped, thus preventing XSS vulnerabilities. At the same time, the code is simple and easy to debug.

The method `response.write(obj, escape=True)` takes two arguments, the object to be written and whether it has to be escaped (set to `True` by default). If `obj` has an `.xml()` method, it is called and the result written to the response body (the `escape` argument is ignored). Otherwise it uses the object's `__str__` method to serialize it and, if the escape argument is `True`, escapes it. All built-in helper objects (`H1` in the example) are objects that know how to serialize themselves via the `.xml()` method.

This is all done transparently. You never need to (and never should) call the `response.write` method explicitly.

## 8.1 Basic syntax

The py4web template language supports all Python control structures. Here we provide some examples of each of them. They can be nested according to usual programming practice.

### 8.1.1 `for...in`

In templates you can loop over any iterable object:

```
[[items = ['a', 'b', 'c']]]
<ul>
[[for item in items:]]<li>[[=item]]</li>[[pass]]
</ul>
```

which produces:

```
<ul>
<li>a</li>
```

```
<li>b</li>
<li>c</li>
</ul>
```

Here `items` is any iterable object such as a Python list, Python tuple, or Rows object, or any object that is implemented as an iterator. The elements displayed are first serialized and escaped.

### 8.1.2 `while`

You can create a loop using the while keyword:

```
[[k = 3]]
<ul>
[[while k > 0:]]<li>[[=k]][[k = k - 1]]</li>[[pass]]
</ul>
```

which produces:

```
<ul>
<li>3</li>
<li>2</li>
<li>1</li>
</ul>
```

### 8.1.3 `if...elif...else`

You can use conditional clauses:

```
[[
import random
k = random.randint(0, 100)
]]
<h2>
[[=k]]
[[if k % 2:]]is odd[[else:]]is even[[pass]]
</h2>
```

which produces:

```
<h2>
45 is odd
</h2>
```

Since it is obvious that `else` closes the first `if` block, there is no need for a `pass` statement, and using one would be incorrect. However, you must explicitly close the `else` block with a `pass`.

Recall that in Python "else if" is written `elif` as in the following example:

```
[[
import random
k = random.randint(0, 100)
]]
<h2>
[[=k]]
[[if k % 4 == 0:]]is divisible by 4
[[elif k % 2 == 0:]]is even
[[else:]]is odd
[[pass]]
```

```
</h2>
```

It produces:

```
<h2>
64 is divisible by 4
</h2>
```

### 8.1.4 try...except...else...finally

It is also possible to use `try...except` statements in views with one caveat. Consider the following example:

```
[[try:]]
Hello [[= 1 / 0]]
[[except:]]
division by zero
[[else:]]
no division by zero
[[finally:]]
<br />
[[pass]]
```

It will produce the following output:

```
Hello division by zero
<br />
```

This example illustrates that all output generated before an exception occurs is rendered (including output that preceded the exception) inside the try block. "Hello" is written because it precedes the exception.

### 8.1.5 def...return

The py4web template language allows the developer to define and implement functions that can return any Python object or a text/html string. Here we consider two examples:

```
[[def itemize1(link): return LI(A(link, _href="http://" + link))]]
<ul>
[[=itemize1('www.google.com')]]
</ul>
```

produces the following output:

```
<ul>
<li><a href="http:/www.google.com">www.google.com</a></li>
</ul>
```

The function `itemize1` returns a helper object that is inserted at the location where the function is called.

Consider now the following code:

```
[[def itemize2(link):]]
<li><a href="http://[[=link]]">[[=link]]</a></li>
[[return]]
<ul>
[[itemize2('www.google.com')]]
```

```
</ul>
```

It produces exactly the same output as above. In this case, the function `itemize2` represents a piece of HTML that is going to replace the py4web tag where the function is called. Notice that there is no '=' in front of the call to `itemize2`, since the function does not return the text, but it writes it directly into the response.

There is one caveat: functions defined inside a view must terminate with a `return` statement, or the automatic indentation will fail.

# YATL helpers

Consider the following code in a view:

```
[[=DIV('this', 'is', 'a', 'test', _id='123', _class='myclass')]]
```

it is rendered as:

```
<div id="123" class="myclass">thisisatest</div>
```

`DIV` is a helper class, i.e., something that can be used to build HTML programmatically. It corresponds to the HTML `<div>` tag.

Positional arguments are interpreted as objects contained between the open and close tags. Named arguments that start with an underscore are interpreted as HTML tag attributes (without the underscore). Some helpers also have named arguments that do not start with underscore; these arguments are tag-specific.

Instead of a set of unnamed arguments, a helper can also take a single list or tuple as its set of components using the `*` notation and it can take a single dictionary as its set of attributes using the `**`, for example:

```
[[
contents = ['this', 'is', 'a', 'test']
attributes = {'_id':'123', '_class':'myclass'}
=DIV(*contents, **attributes)
]]
```

(produces the same output as before).

The following set of helpers:

`A`, `BEAUTIFY`, `BODY`, `CAT`, `CODE`, `DIV`, `EM`, `FORM`, `H1`, `H2`, `H3`, `H4`, `H5`, `H6`, `HEAD`, `HTML`, `I`, `IMG`, `INPUT`, `LABEL`, `LI`, `LINK`, `META`, `METATAG`, `OL`, `OPTION`, `PRE`, `SELECT`, `SPAN`, `STRONG`, `TABLE`, `TAG`, `TBODY`, `TD`, `TEXTAREA`, `TH`, `THEAD`, `TR`, `UL`, `XML`, `sanitize`, `xmlescape`

can be used to build complex expressions that can then be serialized to XML. For example:

```
[[=DIV(B(I("hello ", "<world>")), _class="myclass")]]
```

is rendered:

```
<div class="myclass"><b><i>hello &lt;world&gt;</i></b></div>
```

Helpers can also be serialized into strings, equivalently, with the `__str__` and the `xml` methods:

```
>>> print str(DIV("hello world"))
<div>hello world</div>
>>> print DIV("hello world").xml()
<div>hello world</div>
```

The helpers mechanism in py4web is more than a system to generate HTML without concatenating strings. It provides a server-side representation of the Document Object Model (DOM).

Components of helpers can be referenced via their position, and helpers act as lists with respect to their components:

```
>>> a = DIV(SPAN('a', 'b'), 'c')
>>> print a
<div><span>ab</span>c</div>
>>> del a[1]
>>> a.append(B('x'))
>>> a[0][0] = 'y'
>>> print a
<div><span>yb</span><b>x</b></div>
```

Attributes of helpers can be referenced by name, and helpers act as dictionaries with respect to their attributes:

```
>>> a = DIV(SPAN('a', 'b'), 'c')
>>> a['_class'] = 's'
>>> a[0]['_class'] = 't'
>>> print a
<div class="s"><span class="t">ab</span>c</div>
```

Note, the complete set of components can be accessed via a list called `a.components`, and the complete set of attributes can be accessed via a dictionary called `a.attributes`. So, `a[i]` is equivalent to `a.components[i]` when `i` is an integer, and `a[s]` is equivalent to `a.attributes[s]` when `s` is a string.

Notice that helper attributes are passed as keyword arguments to the helper. In some cases, however, attribute names include special characters that are not allowed in Python identifiers (e.g., hyphens) and therefore cannot be used as keyword argument names. For example:

```
DIV('text', _data-role='collapsible')
```

will not work because "_data-role" includes a hyphen, which will produce a Python syntax error.

In such cases you have a couple of options. You can use the `data` argument (this time without a leading underscore) to pass a dictionary of related attributes without their leading hyphen, and the output will have the desired combinations e.g.

```
>>> print DIV('text', data={'role': 'collapsible'})
<div data-role="collapsible">text</div>
```

or you can instead pass the attributes as a dictionary and make use of Python's `**` function arguments notation, which maps a dictionary of (key:value) pairs into a set of keyword arguments:

```
>>> print DIV('text', **{'_data-role': 'collapsible'})
<div data-role="collapsible">text</div>
```

Note that more elaborate entries will introduce HTML character entities, but they will work nonetheless e.g.

```
>>> print DIV('text', data={'options':'{"mode":"calbox", "useNewStyle":true}'})
<div data-options="{&quot;mode&quot;:&quot;calbox&quot;,
&quot;useNewStyle&quot;:true}">text</div>
```

You can also dynamically create special TAGs:

```
>>> print TAG['soap:Body']('whatever', **{'_xmlns:m':'http://www.example.org'})
<soap:Body xmlns:m="http://www.example.org">whatever</soap:Body>
```

## 9.1 XML

`XML` is an object used to encapsulate text that should not be escaped. The text may or may not contain valid XML. For example, it could contain JavaScript.

The text in this example is escaped:

```
>>> print DIV("<b>hello</b>")
<div>&lt;b&gt;hello&lt;/b&gt;</div>
```

by using `XML` you can prevent escaping:

```
>>> print DIV(XML("<b>hello</b>"))
<div><b>hello</b></div>
```

Sometimes you want to render HTML stored in a variable, but the HTML may contain unsafe tags such as scripts:

```
>>> print XML('<script>alert("unsafe!")</script>')
<script>alert("unsafe!")</script>
```

Un-escaped executable input such as this (for example, entered in the body of a comment in a blog) is unsafe, because it can be used to generate Cross Site Scripting (XSS) attacks against other visitors to the page.

The py4web `XML` helper can sanitize our text to prevent injections and escape all tags except those that you explicitly allow. Here is an example:

```
>>> print XML('<script>alert("unsafe!")</script>', sanitize=True)
&lt;script&gt;alert(&quot;unsafe!&quot;)&lt;/script&gt;
```

The `XML` constructors, by default, consider the content of some tags and some of their attributes safe. You can override the defaults using the optional `permitted_tags` and `allowed_attributes` arguments. Here are the default values of the optional arguments of the `XML` helper.

```
XML(text, sanitize=False,
    permitted_tags=['a', 'b', 'blockquote', 'br/', 'i', 'li',
       'ol', 'ul', 'p', 'cite', 'code', 'pre', 'img/'],
    allowed_attributes={'a':['href', 'title'],
       'img':['src', 'alt'], 'blockquote':['type']})
```

## 9.2 Built-in helpers

### 9.2.1 A

This helper is used to build links.

```
>>> print A('<click>', XML('<b>me</b>'),
            _href='http://www.py4web.com')
<a href='http://www.py4web.com'>&lt;click&gt;<b>me</b></a>
```

### 9.2.2 BODY

This helper makes the body of a page.

```
>>> print BODY('<hello>', XML('<b>world</b>'), _bgcolor='red')
<body bgcolor="red">&lt;hello&gt;<b>world</b></body>
```

### 9.2.3 CAT

This helper concatenates other helpers, same as TAG[''].

```
>>> print CAT('Here is a ', A('link', _href=URL()), ', and here is some ', B('bold
text'), '.')
Here is a <a href="/app/default/index">link</a>, and here is some <b>bold text</b>.
```

### 9.2.4 CODE

This helper performs syntax highlighting for Python, C, C++, HTML and py4web code, and is preferable to PRE for code listings. CODE also has the ability to create links to the py4web API documentation.

Here is an example of highlighting sections of Python code.

```
>>> print CODE('print "hello"', language='python').xml()
```

```
<table><tr style="vertical-align:top;">
  <td style="min-width:40px; text-align: right;"><pre style="
        font-size: 11px;
        font-family: Bitstream Vera Sans Mono,monospace;
        background-color: transparent;
        margin: 0;
        padding: 5px;
        border: none;
        color: #A0A0A0;
    ">1.</pre></td><td><pre style="
        font-size: 11px;
        font-family: Bitstream Vera Sans Mono,monospace;
        background-color: transparent;
        margin: 0;
        padding: 5px;
        border: none;
        overflow: auto;
        white-space: pre !important;
"><span style="color:#185369; font-weight: bold">print </span>
```

```
  <span style="color: #FF9966">"hello"</span></pre></td></tr></table>
```

Here is a similar example for HTML

```
>>> print CODE('<html><body>[[=request.env.remote_add]]</body></html>',
...     language='html')
```

```
<table>...<code>...
<html><body>[[=request.env.remote_add]]</body></html>
...</code>...</table>
```

These are the default arguments for the `CODE` helper:

```
CODE("print 'hello world'", language='python', link=None, counter=1, styles={})
```

Supported values for the `language` argument are "python", "html_plain", "c", "cpp", "py4web", and "html". The "html" language interprets tags as "py4web" code, while "html_plain" doesn't.

If a `link` value is specified, for example "/examples/global/vars/", py4web API references in the code are linked to documentation at the link URL. For example "request" would be linked to "/examples/global/vars/request". In the above example, the link URL is handled by the "vars" action in the "global.py" controller that is distributed as part of the py4web "examples" application.

The `counter` argument is used for line numbering. It can be set to any of three different values. It can be `None` for no line numbers, a numerical value specifying the start number, or a string. If the counter is set to a string, it is interpreted as a prompt, and there are no line numbers.

The `styles` argument is a bit tricky. If you look at the generated HTML above, it contains a table with two columns, and each column has its own style declared inline using CSS. The `styles` attributes allows you to override those two CSS styles. For example:

```
CODE(..., styles={'CODE':'margin: 0;padding: 5px;border: none;'})
```

The `styles` attribute must be a dictionary, and it allows two possible keys: `CODE` for the style of the actual code, and `LINENUMBERS` for the style of the left column, which contains the line numbers. Mind that these styles completely replace the default styles and are not simply added to them.

### 9.2.5 DIV

All helpers apart from `XML` are derived from `DIV` and inherit its basic methods.

```
>>> print DIV('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<div id="0" class="test">&lt;hello&gt;<b>world</b></div>
```

### 9.2.6 EM

Emphasizes its content.

```
>>> print EM('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<em id="0" class="test">&lt;hello&gt;<b>world</b></em>
```

### 9.2.7 FORM

This is one of the most important helpers. In its simple form, it just makes a `<form>...</form>` tag, but because helpers are objects and have knowledge of what they contain, they can process submitted forms (for example, perform validation of the fields). This will be discussed in detail in Chapter 10.

```
>>> print FORM(INPUT(_type='submit'), _action='', _method='post')
<form enctype="multipart/form-data" action="" method="post">
<input type="submit" /></form>
```

The "enctype" is "multipart/form-data" by default.

The constructor of a `FORM`, and of `SQLFORM`, can also take a special argument called `hidden`. When a dictionary is passed as `hidden`, its items are translated into "hidden" INPUT fields. For example:

```
>>> print FORM(hidden=dict(a='b'))
<form enctype="multipart/form-data" action="" method="post">
<input value="b" type="hidden" name="a" /></form>
```

### 9.2.8 H1, H2, H3, H4, H5, H6

These helpers are for paragraph headings and subheadings:

```
>>> print H1('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<h1 id="0" class="test">&lt;hello&gt;<b>world</b></h1>
```

### 9.2.9 HEAD

For tagging the HEAD of an HTML page.

```
>>> print HEAD(TITLE('<hello>', XML('<b>world</b>')))
<head><title>&lt;hello&gt;<b>world</b></title></head>
```

### 9.2.10 HTML

This helper is a little different. In addition to making the `<html>` tags, it prepends the tag with a doctype string.

```
>>> print HTML(BODY('<hello>', XML('<b>world</b>')))
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html><body>&lt;hello&gt;<b>world</b></body></html>
```

The HTML helper also takes some additional optional arguments that have the following default:

```
HTML(..., lang='en', doctype='transitional')
```

where doctype can be 'strict', 'transitional', 'frameset', 'html5', or a full doctype string.

### 9.2.11 I

This helper makes its contents italic.

```
>>> print I('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<i id="0" class="test">&lt;hello&gt;<b>world</b></i>
```

### 9.2.12 IMG

It can be used to embed images into HTML:

```
>>> print IMG(_src='http://example.com/image.png', _alt='test')
```

```
 ![](http://example.com/image.ong){ alt="rest" }
```

Here is a combination of A, IMG, and URL helpers for including a static image with a link:

```
>>> print A(IMG(_src=URL('static', 'logo.png'), _alt="My Logo"),
...   _href=URL('default', 'index'))
...
<a href="/myapp/default/index">
   ![](/myapp/static/logo.png){ alt="My Logo" }
</a>
```

### 9.2.13 INPUT

Creates an `<input.../>` tag. An input tag may not contain other tags, and is closed by `/>` instead of `>`. The input tag has an optional attribute `_type` that can be set to "text" (the default), "submit", "checkbox", or "radio".

```
>>> print INPUT(_name='test', _value='a')
<input value="a" name="test" />
```

It also takes an optional special argument called "value", distinct from "_value". The latter sets the default value for the input field; the former sets its current value. For an input of type"text", the former overrides the latter:

```
>>> print INPUT(_name='test', _value='a', value='b')
<input value="b" name="test" />
```

For radio buttons, `INPUT` selectively sets the "checked" attribute:

```
>>> for v in ['a', 'b', 'c']:
...     print INPUT(_type='radio', _name='test', _value=v, value='b'), v
...
<input value="a" type="radio" name="test" /> a
<input value="b" type="radio" checked="checked" name="test" /> b
<input value="c" type="radio" name="test" /> c
```

and similarly for checkboxes:

```
>>> print INPUT(_type='checkbox', _name='test', _value='a', value=True)
<input value="a" type="checkbox" checked="checked" name="test" />
>>> print INPUT(_type='checkbox', _name='test', _value='a', value=False)
<input value="a" type="checkbox" name="test" />
```

### 9.2.14 LABEL

It is used to create a LABEL tag for an INPUT field.

```
>>> print LABEL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<label id="0" class="test">&lt;hello&gt;<b>world</b></label>
```

### 9.2.15 LI

It makes a list item and should be contained in a `UL` or `OL` tag.

```
>>> print LI('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<li id="0" class="test">&lt;hello&gt;<b>world</b></li>
```

### 9.2.16 OL

It stands for Ordered List. The list should contain LI tags. OL arguments that are not LI objects are automatically enclosed in <li>...</li> tags.

```
>>> print OL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<ol id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></ol>
```

### 9.2.17 OPTION

This should only be used as part of a SELECT/OPTION combination.

```
>>> print OPTION('<hello>', XML('<b>world</b>'), _value='a')
<option value="a">&lt;hello&gt;<b>world</b></option>
```

As in the case of INPUT, py4web make a distinction between "_value" (the value of the OPTION), and "value" (the current value of the enclosing select). If they are equal, the option is "selected".

```
>>> print SELECT('a', 'b', value='b'):
<select>
<option value="a">a</option>
<option value="b" selected="selected">b</option>
</select>
```

### 9.2.18 P

This is for tagging a paragraph.

```
>>> print P('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<p id="0" class="test">&lt;hello&gt;<b>world</b></p>
```

### 9.2.19 PRE

Generates a <pre>...</pre> tag for displaying pre-formatted text. The CODE helper is generally preferable for code listings.

```
>>> print PRE('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<pre id="0" class="test">&lt;hello&gt;<b>world</b></pre>
```

### 9.2.20 SCRIPT

This is include or link a script, such as JavaScript. The content between the tags is rendered as an HTML comment, for the benefit of really old browsers.

```
>>> print SCRIPT('alert("hello world");', _type='text/javascript')
<script type="text/javascript"><!--
alert("hello world");
//--></script>
```

### 9.2.21 SELECT

Makes a <select>...</select> tag. This is used with the OPTION helper. Those SELECT arguments that are not OPTION objects are automatically converted to options.

```
>>> print SELECT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<select id="0" class="test">
<option value="&lt;hello&gt;">&lt;hello&gt;</option>
<option value="&lt;b&gt;world&lt;/b&gt;"><b>world</b></option>
</select>
```

### 9.2.22 SPAN

Similar to `DIV` but used to tag inline (rather than block) content.

```
>>> print SPAN('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<span id="0" class="test">&lt;hello&gt;<b>world</b></span>
```

### 9.2.23 STYLE

Similar to script, but used to either include or link CSS code. Here the CSS is included:

```
>>> print STYLE(XML('body {color: white}'))
<style><!--
body { color: white }
//--></style>
```

and here it is linked:

```
>>> print STYLE(_src='style.css')
<style src="style.css"><!--
//--></style>
```

### 9.2.24 TABLE, TR, TD

These tags (along with the optional `THEAD` and `TBODY` helpers) are used to build HTML tables.

```
>>> print TABLE(TR(TD('a'), TD('b')), TR(TD('c'), TD('d')))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

`TR` expects `TD` content; arguments that are not `TD` objects are converted automatically.

```
>>> print TABLE(TR('a', 'b'), TR('c', 'd'))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

It is easy to convert a Python array into an HTML table using Python's `*` function arguments notation, which maps list elements to positional function arguments.

Here, we will do it line by line:

```
>>> table = [['a', 'b'], ['c', 'd']]
>>> print TABLE(TR(*table[0]), TR(*table[1]))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

Here we do all lines at once:

```
>>> table = [['a', 'b'], ['c', 'd']]
>>> print TABLE(*[TR(*rows) for rows in table])
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

### 9.2.25 TBODY

This is used to tag rows contained in the table body, as opposed to header or footer rows. It is optional.

```
>>> print TBODY(TR('<hello>'), _class='test', _id=0)
<tbody id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tbody>
```

### 9.2.26 TEXTAREA

This helper makes a `<textarea>...</textarea>` tag.

```
>>> print TEXTAREA('<hello>', XML('<b>world</b>'), _class='test')
<textarea class="test" cols="40" rows="10">&lt;hello&gt;<b>world</b></textarea>
```

The only caveat is that its optional "value" overrides its content (inner HTML)

```
>>> print TEXTAREA(value="<hello world>", _class="test")
<textarea class="test" cols="40" rows="10">&lt;hello world&gt;</textarea>
```

### 9.2.27 TH

This is used instead of `TD` in table headers.

```
>>> print TH('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<th id="0" class="test">&lt;hello&gt;<b>world</b></th>
```

### 9.2.28 THEAD

This is used to tag table header rows.

```
>>> print THEAD(TR(TH('<hello>')), _class='test', _id=0)
<thead id="0" class="test"><tr><th>&lt;hello&gt;</th></tr></thead>
```

### 9.2.29 TITLE

This is used to tag the title of a page in an HTML header.

```
>>> print TITLE('<hello>', XML('<b>world</b>'))
<title>&lt;hello&gt;<b>world</b></title>
```

### 9.2.30 TR

Tags a table row. It should be rendered inside a table and contain `<td>...</td>` tags. `TR` arguments that are not `TD` objects will be automatically converted.

```
>>> print TR('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<tr id="0" class="test"><td>&lt;hello&gt;</td><td><b>world</b></td></tr>
```

### 9.2.31 TT

Tags text as typewriter (monospaced) text.

```
>>> print TT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<tt id="0" class="test">&lt;hello&gt;<b>world</b></tt>
```

### 9.2.32 UL

Signifies an Unordered List and should contain LI items. If its content is not tagged as LI, UL does it automatically.

```
>>> print UL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<ul id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></ul>
```

### 9.2.33 URL

The URL helper is documented in *Chapter 4 URL ../04*

## 9.3 Custom helpers

### 9.3.1 TAG

Sometimes you need to generate custom XML tags. py4web provides TAG, a universal tag generator.

```
[[=TAG.name('a', 'b', _c='d')]]
```

generates the following XML

```
<name c="d">ab</name>
```

Arguments "a", "b", and "d" are automatically escaped; use the XML helper to suppress this behavior. Using TAG you can generate HTML/XML tags not already provided by the API. TAGs can be nested, and are serialized with str(). An equivalent syntax is:

```
[[=TAG['name']('a', 'b', c='d')]]
```

If the TAG object is created with an empty name, it can be used to concatenate multiple strings and HTML helpers together without inserting them into a surrounding tag, but this use is deprecated. Use the CAT helper instead.

Self-closing tags can be generated with the TAG helper. The tag name must end with a "/".

```
[[=TAG['link/'](_href='http://py4web.com')]]
```

generates the following XML:

```
<link ref="http://py4web.com"/>
```

Notice that TAG is an object, and TAG.name or TAG['name'] is a function that returns a temporary helper class.

### 9.3.2 MENU

The MENU helper takes a list of lists or of tuples of the form of response.menu and generates a tree-like structure using unordered lists representing the menu. For example:

```
>>> print MENU([['One', False, 'link1'], ['Two', False, 'link2']])
<ul class="py4web-menu py4web-menu-vertical">
<li><a href="link1">One</a></li>
<li><a href="link2">Two</a></li>
```

```
</ul>
```

The first item in each list/tuple is the text to be displayed for the given menu item.

The second item in each list/tuple is a boolean indicating whether that particular menu item is active (i.e., the currently selected item). When set to True, the `MENU` helper will add a "py4web-menu-active" class to the `<li>` for that item (you can change the name of that class via the "li_active" argument to `MENU`). Another way to specify the active url is by directly passing it to `MENU` via its "active_url" argument.

The third item in each list/tuple can be an HTML helper (which could include nested helpers), and the `MENU` helper will simply render that helper rather than creating its own `<a>` tag.

Each menu item can have a fourth argument that is a nested submenu (and so on recursively):

```
>>> print MENU([['One', False, 'link1', [['Two', False, 'link2']]]])
<ul class="py4web-menu py4web-menu-vertical">
<li class="py4web-menu-expand">
<a href="link1">One</a>
<ul class="py4web-menu-vertical">
<li><a href="link2">Two</a></li>
</ul>
</li>
</ul>
```

A menu item can also have an optional 5th element, which is a boolean. When false, the menu item is ignored by the MENU helper.

The `MENU` helper takes the following optional arguments: - `_class`: defaults to "py4web-menu py4web-menu-vertical" and sets the class of the outer UL elements. - `ul_class`: defaults to "py4web-menu-vertical" and sets the class of the inner UL elements. - `li_class`: defaults to "py4web-menu-expand" and sets the class of the inner LI elements. - `li_first`: allows to add a class to the first list element. - `li_last`: allows to add a class to the last list element.

`MENU` takes an optional argument `mobile`. When set to `True` instead of building a recursive `UL` menu structure it returns a `SELECT` dropdown with all the menu options and a `onchange` attribute that redirects to the page corresponding to the selected option. This is designed an an alternative menu representation that increases usability on small mobile devices such as phones.

Normally the menu is used in a layout with the following syntax:

```
[[=MENU(response.menu, mobile=request.user_agent().is_mobile)]]
```

In this way a mobile device is automatically detected and the menu is rendered accordingly.

## 9.4 BEAUTIFY

`BEAUTIFY` is used to build HTML representations of compound objects, including lists, tuples and dictionaries:

```
[[=BEAUTIFY({"a": ["hello", XML("world")], "b": (1, 2)})]]
```

`BEAUTIFY` returns an XML-like object serializable to XML, with a nice looking representation of its constructor argument. In this case, the XML representation of:

```
{"a": ["hello", XML("world")], "b": (1, 2)}
```

will render as:

```
<table>
<tr><td>a</td><td>:</td><td>hello<br />world</td></tr>
<tr><td>b</td><td>:</td><td>1<br />2</td></tr>
</table>
```

## 9.5 Server-side *DOM* and parsing

### 9.5.1 `elements`

The DIV helper and all derived helpers provide the search methods `element` and `elements`.

`element` returns the first child element matching a specified condition (or None if no match).

`elements` returns a list of all matching children.

**element** and **elements** use the same syntax to specify the matching condition, which allows for three possibilities that can be mixed and matched: jQuery-like expressions, match by exact attribute value, match using regular expressions.

Here is a simple example:

```
>>> a = DIV(DIV(DIV('a', _id='target', _class='abc')))
>>> d = a.elements('div#target')
>>> d[0][0] = 'changed'
>>> print a
<div><div><div id="target" class="abc">changed</div></div></div>
```

The un-named argument of `elements` is a string, which may contain: the name of a tag, the id of a tag preceded by a pound symbol, the class preceded by a dot, the explicit value of an attribute in square brackets.

Here are 4 equivalent ways to search the previous tag by id:

```
d = a.elements('#target')
d = a.elements('div#target')
d = a.elements('div[id=target]')
d = a.elements('div', _id='target')
```

Here are 4 equivalent ways to search the previous tag by class:

```
d = a.elements('.abc')
d = a.elements('div.abc')
d = a.elements('div[class=abc]')
d = a.elements('div', _class='abc')
```

Any attribute can be used to locate an element (not just `id` and `class`), including multiple attributes (the function element can take multiple named arguments), but only the first matching element will be returned.

Using the jQuery syntax "div#target" it is possible to specify multiple search criteria separated by a comma:

```
a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))
d = a.elements('span#t1, div.c2')
```

or equivalently

```
a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))
d = a.elements('span#t1', 'div.c2')
```

If the value of an attribute is specified using a name argument, it can be a string or a regular expression:

```
a = DIV(SPAN('a', _id='test123'), DIV('b', _class='c2'))
d = a.elements('span', _id=re.compile('test\d{3}')
```

A special named argument of the DIV (and derived) helpers is `find`. It can be used to specify a search value or a search regular expression in the text content of the tag. For example:

```
>>> a = DIV(SPAN('abcde'), DIV('fghij'))
>>> d = a.elements(find='bcd')
>>> print d[0]
<span>abcde</span>
```

or

```
>>> a = DIV(SPAN('abcde'), DIV('fghij'))
>>> d = a.elements(find=re.compile('fg\w{3}'))
>>> print d[0]
<div>fghij</div>
```

### 9.5.2 components

Here's an example of listing all elements in an html string:

```
>>> html = TAG('<a>xxx</a><b>yyy</b>')
>>> for item in html.components:
...     print item
...
<a>xxx</a>
<b>yyy</b>
```

### 9.5.3 parent and siblings

`parent` returns the parent of the current element.

```
>>> a = DIV(SPAN('a'), DIV('b'))
>>> s = a.element('span')
>>> d = s.parent
>>> d['_class']='abc'
>>> print a
<div class="abc"><span>a</span><div>b</div></div>
>>> for e in s.siblings(): print e
<div>b</div>
```

### 9.5.4 Replacing elements

Elements that are matched can also be replaced or removed by specifying the `replace` argument. Notice that a list of the original matching elements is still returned as usual.

```
>>> a = DIV(SPAN('x'), DIV(SPAN('y'))
>>> b = a.elements('span', replace=P('z')
>>> print a
<div><p>z</p><div><p>z</p></div>
```

`replace` can be a callable. In this case it will be passed the original element and it is expected to return the replacement element:

```
>>> a = DIV(SPAN('x'), DIV(SPAN('y'))
>>> b = a.elements('span', replace=lambda t: P(t[0])
>>> print a
<div><p>x</p><div><p>y</p></div>
```

If `replace=None`, matching elements will be removed completely.

```
>>> a = DIV(SPAN('x'), DIV(SPAN('y'))
>>> b = a.elements('span', replace=None)
>>> print a
<div></div>
```

### 9.5.5 `flatten`

The flatten method recursively serializes the content of the children of a given element into regular text (without tags):

```
>>> a = DIV(SPAN('this', DIV('is', B('a'))), SPAN('test'))
>>> print a.flatten()
thisisatest
```

Flatten can be passed an optional argument, `render`, i.e. a function that renders/flattens the content using a different protocol. Here is an example to serialize some tags into Markmin wiki syntax:

```
>>> a = DIV(H1('title'), P('example of a ', A('link', _href='#test')))
>>> from gluon.html import markmin_serializer
>>> print a.flatten(render=markmin_serializer)
# titles

example of *a link *
```

At the time of writing we provide `markmin_serializer` and `markdown_serializer`.

### 9.5.6 Parsing

The TAG object is also an XML/HTML parser. It can read text and convert into a tree structure of helpers. This allows manipulation using the API above:

```
>>> html = '<h1>Title</h1><p>this is a <span>test</span></p>'
>>> parsed_html = TAG(html)
>>> parsed_html.element('span')[0]='TEST'
>>> print parsed_html
<h1>Title</h1><p>this is a <span>TEST</span></p>
```

## 9.6 Page layout

Views can extend and include other views in a tree-like structure.

For example, we can think of a view "index.html" that extends "layout.html" and includes "body.html". At the same time, "layout.html" may include "header.html" and "footer.html".

The root of the tree is what we call a layout view. Just like any other HTML template file, you can edit it

using the py4web administrative interface. The file name "layout.html" is just a convention.

Here is a minimalist page that extends the "layout.html" view and includes the "page.html" view:

```
[[extend 'layout.html']]
<h1>Hello World</h1>
[[include 'page.html']]
```

The extended layout file must contain an `[[include]]` directive, something like:

```
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    [[include]]
  </body>
</html>
```

When the view is called, the extended (layout) view is loaded, and the calling view replaces the `[[include]]` directive inside the layout. Processing continues recursively until all `extend` and `include` directives have been processed. The resulting template is then translated into Python code. Note, when an application is bytecode compiled, it is this Python code that is compiled, not the original view files themselves. So, the bytecode compiled version of a given view is a single .pyc file that includes the Python code not just for the original view file, but for its entire tree of extended and included views.

> `extend`, `include`, `block` and `super` are special template directives, not Python commands.

Any content or code that precedes the `[[extend ...]]` directive will be inserted (and therefore executed) before the beginning of the extended view's content/code. Although this is not typically used to insert actual HTML content before the extended view's content, it can be useful as a means to define variables or functions that you want to make available to the extended view. For example, consider a view "index.html":

```
[[sidebar_enabled=True]]
[[extend 'layout.html']]
<h1>Home Page</h1>
```

and an excerpt from "layout.html":

```
[[if sidebar_enabled:]]
    <div id="sidebar">
        Sidebar Content
    </div>
[[pass]]
```

Because the `sidebar_enabled` assignment in "index.html" comes before the `extend`, that line gets inserted before the beginning of "layout.html", making `sidebar_enabled` available anywhere within the "layout.html" code (a somewhat more sophisticated version of this is used in the **welcome** app).

It is also worth pointing out that the variables returned by the controller function are available not only in the function's main view, but in all of its extended and included views as well.

The argument of an `extend` or `include` (i.e., the extended or included view name) can be a Python variable (though not a Python expression). However, this imposes a limitation – views that use variables in `extend` or `include` statements cannot be bytecode compiled. As noted above, bytecode-compiled views include the entire tree of extended and included views, so the specific extended and included views must be known at compile time, which is not possible if the view names are variables (whose values are not determined until run time). Because bytecode compiling views can provide a significant speed boost,

using variables in `extend` and `include` should generally be avoided if possible.

In some cases, an alternative to using a variable in an `include` is simply to place regular `[[include ...]]` directives inside an `if...else` block.

```
[[if some_condition:]]
[[include 'this_view.html']]
[[else:]]
[[include 'that_view.html']]
[[pass]]
```

The above code does not present any problem for bytecode compilation because no variables are involved. Note, however, that the bytecode compiled view will actually include the Python code for both "this_view.html" and "that_view.html", though only the code for one of those views will be executed, depending on the value of `some_condition`.

Keep in mind, this only works for `include` – you cannot place `[[extend ...]]` directives inside `if...else` blocks.

Layouts are used to encapsulate page commonality (headers, footers, menus), and though they are not mandatory, they will make your application easier to write and maintain. In particular, we suggest writing layouts that take advantage of the following variables that can be set in the controller. Using these well known variables will help make your layouts interchangeable:

```
response.title
response.subtitle
response.meta.author
response.meta.keywords
response.meta.description
response.flash
response.menu
response.files
```

Except for `menu` and `files`, these are all strings and their meaning should be obvious.

`response.menu` menu is a list of 3-tuples or 4-tuples. The three elements are: the link name, a boolean representing whether the link is active (is the current link), and the URL of the linked page. For example:

```
response.menu = [('Google', False, 'http://www.google.com', []),
                 ('Index',  True,  URL('index'), [])]
```

The fourth tuple element is an optional sub-menu.

`response.files` is a list of CSS and JS files that are needed by your page.

We also recommend that you use:

```
[[include 'py4web_ajax.html']]
```

in the HTML head, since this will include the jQuery libraries and define some backward-compatible JavaScript functions for special effects and Ajax. "py4web_ajax.html" includes the `response.meta` tags in the view, jQuery base, the calendar datepicker, and all required CSS and JS `response.files`.

### 9.6.1 Default page layout

The "views/layout.html" that ships with the py4web scaffolding application **welcome** (stripped down of some optional parts) is quite complex but it has the following structure:

```
<!DOCTYPE html>
<head>
```

```
  <meta charset="utf-8" />
  <title>[[=response.title or request.application]]</title>
  ...
  <script src="[[=URL('static', 'js/modernizr.custom.js')]]"></script>

  [[
  response.files.append(URL('static', 'css/py4web.css'))
  response.files.append(URL('static', 'css/bootstrap.min.css'))
  response.files.append(URL('static', 'css/bootstrap-responsive.min.css'))
  response.files.append(URL('static', 'css/py4web_bootstrap.css'))
  ]]

  [[include 'py4web_ajax.html']]

  [[
  # using sidebars need to know what sidebar you want to use
  left_sidebar_enabled = globals().get('left_sidebar_enabled', False)
  right_sidebar_enabled = globals().get('right_sidebar_enabled', False)
  middle_columns = {0:'span12', 1:'span9', 2:'span6'}[
    (left_sidebar_enabled and 1 or 0)+(right_sidebar_enabled and 1 or 0)]
  ]]

  [[block head]][[end]]
</head>

<body>
  <!-- Navbar ================================================== -->
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="flash">[[=response.flash or '']]</div>
    <div class="navbar-inner">
      <div class="container">
        [[=response.logo or '']]
        <ul id="navbar" class="nav pull-right">
          [['auth' in globals() and auth.navbar(mode="dropdown") or '']]
        </ul>
        <div class="nav-collapse">
          [[if response.menu:]]
          [[=MENU(response.menu)]]
          [[pass]]
        </div><!--/.nav-collapse -->
      </div>
    </div>
  </div><!--/top navbar -->

  <div class="container">
    <!-- Masthead ================================================== -->
    <header class="mastheader row" id="header">
        <div class="span12">
            <div class="page-header">
                <h1>
                    [[=response.title or request.application]]
                    <small>[[=response.subtitle or '']]</small>
                </h1>
            </div>
        </div>
    </header>

    <section id="main" class="main row">
        [[if left_sidebar_enabled:]]
```

```
        <div class="span3 left-sidebar">
            [[block left_sidebar]]
            <h3>Left Sidebar</h3>
            <p></p>
            [[end]]
        </div>
        [[pass]]

        <div class="[[=middle_columns]]">
            [[block center]]
            [[include]]
            [[end]]
        </div>

        [[if right_sidebar_enabled:]]
        <div class="span3">
            [[block right_sidebar]]
            <h3>Right Sidebar</h3>
            <p></p>
            [[end]]
        </div>
        [[pass]]
    </section><!--/main-->

    <!-- Footer ================================================== -->
    <div class="row">
        <footer class="footer span12" id="footer">
            <div class="footer-content">
                [[block footer]] <!-- this is default footer -->
                ...
                [[end]]
            </div>
        </footer>
    </div>

  </div> <!-- /container -->

  <!-- The javascript ============================================
       (Placed at the end of the document so the pages load faster) -->
  <script src="[[=URL('static', 'js/bootstrap.min.js')]]"></script>
  <script src="[[=URL('static', 'js/py4web_bootstrap.js')]]"></script>
  [[if response.google_analytics_id:]]
    <script src="[[=URL('static', 'js/analytics.js')]]"></script>
    <script type="text/javascript">
    analytics.initialize({
      'Google Analytics':{trackingId:'[[=response.google_analytics_id]]'}
    });</script>
  [[pass]]
</body>
</html>
```

There are a few features of this default layout that make it very easy to use and customize:

- It is written in HTML5 and uses the "modernizr" library for backward compatibility. The actual layout includes some extra conditional statements required by IE and they are omitted for brevity.
- It displays both `response.title` and `response.subtitle` which can be set in a model or a controller. If they are not set, it adopts the application name as title.
- It includes the `py4web_ajax.html` file in the header which generated all the link and script import

statements.

- It uses a modified version of Twitter Bootstrap for flexible layouts which works on mobile devices and re-arranges columns to fit small screens.
- It uses "analytics.js" to connect to Google Analytics.
- The `[[=auth.navbar(...)]]` displays a welcome to the current user and links to the auth functions like login, logout, register, change password, etc. depending on context. `auth.navbar` is a helper factory and its output can be manipulated as any other helper. It is placed in an expression to check for auth definition, the expression evaluates to '' in case auth is undefined.
- The `[[=MENU(response.menu)]]` displays the menu structure as `<ul>...</ul>`.
- `[[include]]` is replaced by the content of the extending view when the page is rendered.
- By default it uses a conditional three column (the left and right sidebars can be turned off by the extending views)
- It uses the following classes: page-header, main, footer.
- It contains the following blocks: head, left_sidebar, center, right_sidebar, footer.

In views, you can turn on and customize sidebars as follows:

```
[[left_sidebar_enabled=True]]
[[extend 'layout.html']]

This text goes in center

[[block left_sidebar]]
This text goes in sidebar
[[end]]
```

### 9.6.2 Customizing the default layout

Customizing the default layout without editing is easy because the welcome application is based on Twitter Bootstrap which is well documented and supports themes. In py4web four static files which are relevant to style:

- "css/py4web.css" contains py4web specific styles
- "css/bootstrap.min.css" contains the Twitter Bootstrap CSS style
- "css/py4web_bootstrap.css" which overrides some Bootstrap styles to conform to py4web needs.
- "js/bootstrap.min.js" which includes the libraries for menu effects, modals, panels.

To change colors and background images, try append the following code to layout.html header:

```
<style>
body { background: url('images/background.png') repeat-x #3A3A3A; }
a { color: #349C01; }
.page-header h1 { color: #349C01; }
.page-header h2 { color: white; font-style: italic; font-size: 14px;}
.statusbar { background: #333333; border-bottom: 5px #349C01 solid; }
.statusbar a { color: white; }
.footer { border-top: 5px #349C01 solid; }
</style>
```

Of course you can also completely replace the "layout.html" and "py4web.css" files with your own.

### 9.6.3 Mobile development

Although the default layout.html is designed to be mobile-friendly, one may sometimes need to use different views when a page is visited by a mobile device.

To make developing for desktop and mobile devices easier, py4web includes the `@mobilize` decorator. This decorator is applied to actions that should have a normal view and a mobile view. This is demonstrated here:

```
from gluon.contrib.user_agent_parser import mobilize
@mobilize
def index():
    return dict()
```

Notice that the decorator must be imported before using it in a controller. When the "index" function is called from a regular browser (desktop computer), py4web will render the returned dictionary using the view "[controller]/index.html". However, when it is called by a mobile device, the dictionary will be rendered by "[controller]/index.mobile.html". Notice that mobile views have the "mobile.html" extension.

Alternatively you can apply the following logic to make all views mobile friendly:

```
if request.user_agent().is_mobile:
    response.view.replace('.html', '.mobile.html')
```

The task of creating the "*.mobile.html" views is left to the developer but we strongly suggest using the "jQuery Mobile" plugin which makes the task very easy.

## 9.7 Functions in views

Consider this "layout.html":

```
<html>
  <body>
    [[include]]
    <div class="sidebar">
      [[if 'mysidebar' in globals():]][[mysidebar()]][[else:]]
        my default sidebar
      [[pass]]
    </div>
  </body>
</html>
```

and this extending view

```
[[def mysidebar():]]
my new sidebar!!!
[[return]]
[[extend 'layout.html']]
Hello World!!!
```

Notice the function is defined before the `[[extend...]]` statement – this results in the function being created before the "layout.html" code is executed, so the function can be called anywhere within "layout.html", even before the `[[include]]`. Also notice the function is included in the extended view without the = prefix.

The code generates the following output:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
      my new sidebar!!!
    </div>
  </body>
</html>
```

Notice that the function is defined in HTML (although it could also contain Python code) so that `response.write` is used to write its content (the function does not return the content). This is why the layout calls the view function using `[[mysidebar()]]` rather than `[[=mysidebar()]]`. Functions defined in this way can take arguments.

## 9.8 Blocks in views

The main way to make a view more modular is by using `[[block ...]]`s and this mechanism is an alternative to the mechanism discussed in the previous section.

To understand how this works, consider apps based on the scaffolding app welcome, which has a view layout.html. This view is extended by the view `default/index.html` via `[[extend 'layout.html']]`. The contents of layout.html predefine certain blocks with certain default content, and these are therefore included into default/index.html.

You can override these default content blocks by enclosing your new content inside the same block name. The location of the block in the layout.html is not changed, but the contents is.

Here is a simplifed version. Imagine this is "layout.html":

```
<html>
  <body>
    [[include]]
    <div class="sidebar">
      [[block mysidebar]]
        my default sidebar (this content to be replaced)
      [[end]]
    </div>
  </body>
</html>
```

and this is a simple extending view `default/index.html`:

```
[[extend 'layout.html']]
Hello World!!!
[[block mysidebar]]
my new sidebar!!!
[[end]]
```

It generates the following output, where the content is provided by the over-riding block in the extending view, yet the enclosing DIV and class comes from layout.html. This allows consistency across views:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
```

```
        my new sidebar!!!
    </div>
  </body>
</html>
```

The real layout.html defines a number of useful blocks, and you can easily add more to match the layout your desire.

You can have many blocks, and if a block is present in the extended view but not in the extending view, the content of the extended view is used. Also, notice that unlike with functions, it is not necessary to define blocks before the `[[extend ...]]` – even if defined after the `extend`, they can be used to make substitutions anywhere in the extended view.

Inside a block, you can use the expression `[[super]]` to include the content of the parent. For example, if we replace the above extending view with:

```
[[extend 'layout.html']]
Hello World!!!
[[block mysidebar]]
[[super]]
my new sidebar!!!
[[end]]
```

we get:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
        my default sidebar
        my new sidebar!
    </div>
  </body>
</html>
```

Chapter 10

# Internationalization

## 10.1 Pluralize

Pluralize is a Python library for Internationalization (i18n) and Pluralization (p10n).

The library assumes a folder (for exaple "translations") that contains files like:

```
it.json
it-IT.json
fr.json
fr-FR.json
(etc)
```

Each file has the following structure, for example for Italian (it.json):

```
{"dog": {"0": "no cane", "1": "un cane", "2": "{n} cani", "10": "tantissimi cani"}}
```

The top level keys are the expressions to be translated and the associated value/dictionary maps a number to a translation. Different translations correspond to different plural forms of the expression,

Here is another example for the word "bed" in Czech

```
{"bed": {"0": "no postel", "1": "postel", "2": "postele", "5": "postelí"}}
```

To translate and pluralize a string "dog" one simply warps the string in the T operator as follows:

```
>>> from pluralize import Translator
>>> T = Translator('translations')
>>> dog = T("dog")
>>> print(dog)
dog
>>> T.select('it')
>>> print(dog)
un cane
>>> print(dog.format(n=0))
no cane
>>> print(dog.format(n=1))
un cane
>>> print(dog.format(n=5))
5 cani
>>> print(dog.format(n=20))
```

```
tantissimi cani
```

The string can contain multiple placeholders but the {n} placeholder is special because the variable called "n" is used to determine the pluralization by best match (max dict key <= n).

T(…) objects can be added together with each other and with string, like regular strings.

T.select(s) can parse a string s following the HTTP accept language format.

## 10.2 Update the translation files

Find all strings wrapped in T(…) in .py, .html, and .js files:

```
matches = T.find_matches('path/to/app/folder')
```

Add newly discovered entries in all supported languages

```
T.update_languages(matches)
```

Add a new supported language (for example german, "de")

```
T.languages['de'] = {}
```

Make sure all languages contain the same origin expressions

```
known_expressions = set()
for language in T.languages.values():
    for expression in language:
        known_expressions.add(expression)
T.update_languages(known_expressions))
```

Finally save the changes:

```
T.save('translations')
```

Chapter 11

# Forms

WORK IN PROGRESS

Just know that `py4web.utils.form.Form` is a pretty much equivalent to web2py's `SQLFORM`.

The `Form` constructor accepts the following arguments:

```
Form(self,
     table,
     record=None,
     readonly=False,
     deletable=True,
     formstyle=FormStyleDefault,
     dbio=True,
     keep_values=False,
     form_name=False,
     hidden=None,
     before_validate=None):
```

Where:

- `table`: a DAL table or a list of fields (equivalent to old SQLFORM.factory)
- `record`: a DAL record or record id
- `readonly`: set to True to make a readonly form
- `deletable`: set to False to disallow deletion of record
- `formstyle`: a function that renders the form using helpers (FormStyleDefault)
- `dbio`: set to False to prevent any DB writes
- `keep_values`: if set to true, it remembers the values of the previously submitted form
- `form_name`: the optional name of this form
- `hidden`: a dictionary of hidden fields that is added to the form
- `before_validate`: an optional validator.

## 11.1 Example

Here is a simple example of a custom form not using database access. We declare an endpoint `/form_example`, which will be used both for the GET and for the POST of the form:

```
from py4web import Session, redirect, URL
from py4web.utils.dbstore import DBStore
from py4web.utils.form import Form, FormStyleBulma

db = DAL('sqlite:memory')
session =  Session(storage=DBStore(db))

@action('form_example', method=['GET', 'POST'])
@action.uses('form_example.html', session)
def form_example():
    form = Form([
        Field('product_name'),
        Field('product_quantity', 'integer')],
        formstyle=FormStyleBulma)
    if form.accepted:
        # Do something with form.vars['product_name'] and
form.vars['product_quantity']
        redirect(URL('index'))
    return dict(form=form)
```

The form can be displayed in the template simply using `[[=form]]`.

## 11.2 Form validation

The validation of form input can be done in two ways. One can define `requires` attributes of `Field`, or one can define explicitly a validation function. To do the latter, we pass to `validate` a function that takes the form and returns a dictionary, mapping field names to errors. If the dictionary is non-empty, the errors will be displayed to the user, and no database I/O will take place.

Here is an example:

```
from py4web import Field
from py4web.utils.form import Form, FormStyleBulma
from pydal.validators import IS_INT_IN_RANGE

def check_nonnegative_quantity(form):
    if not form.errors and form.vars['product_quantity'] % 2:
        form.errors['product_quantity'] = T('The product quantity must be even')

@action('form_example', method=['GET', 'POST'])
@action.uses('form_example.html', session)
def form_example():
    form = Form([
        Field('product_name'),
        Field('product_quantity', 'integer', requires=IS_INT_IN_RANGE(0,100))],
        validation=check_nonnegative_quantity,
        formstyle=FormStyleBulma)
    if form.accepted:
        # Do something with form.vars['product_name'] and
form.vars['product_quantity']
        redirect(URL('index'))
    return dict(form=form)
```

Chapter 12

# Authentication and Access control

**Warning: the API described in this chapter is new and subject to changes. Make sure you keep your code up to date**

py4web comes with a an object Auth and a system of plugins for user authentication and access control. It has the same name as the corresponding web2py one and serves the same purpose but the API and internal design is very different.

To use it, first of all you need to import it, instantiate it, configure it, and enable it.

```
from py4web.utils.auth import Auth
auth = Auth(session, db)
# (configure here)
auth.enable()
```

The import step is obvious. The second step does not perform any operation other than telling the Auth object which session object to use and which database to use. Auth data is stored in `session['user']` and, if a user is logged in, the user id is stored in session['user']['id']. The db object is used to store persistent info about the user in a table `auth_user` with the following fields:

- username
- email
- password
- first_name
- last_name
- sso_id (used for single sign on, see later)
- action_token (used to verify email, block users, and other tasks, also see later).

If the `auth_user` table does not exist it is created.

The configuration step is optional and discussed later.

The `auth.enable()` step creates and exposes the following RESTful APIs:

- {appname}/auth/api/register (POST)
- {appname}/auth/api/login (POST)
- {appname}/auth/api/request_reset_password (POST)
- {appname}/auth/api/reset_password (POST)
- {appname}/auth/api/verify_email (GET, POST)

- {appname}/auth/api/logout (GET, POST) (+)
- {appname}/auth/api/profile (GET, POST) (+)
- {appname}/auth/api/change_password (POST) (+)
- {appname}/auth/api/change_email (POST) (+)

Those marked with a (+) require a logged in user.

## 12.1 Auth UI

You can create your own web UI to login users using the above APIs but py4web provides one as an example, implemented in the following files:

- _scaffold/templates/auth.html
- _scaffold/static/components/auth.js
- _scaffold/static/components/auth.html

The component files (js/html) define a Vue component `<auth/>` which is used in the template file auth.html as follows:

```
[[extend "layout.html"]]
<div id="vue">
  <div class="columns">
    <div class="column is-half is-offset-one-quarter" style="border : 1px solid
#e1e1e1; border-radius: 10px">
      <auth plugins="local,oauth2google,oauth2facebook"></auth>
    </div>
  </div>
</div>
[[block page_scripts]]
<script src="js/utils.js"></script>
<script src="components/auth.js"></script>
<script>utils.app().start();</script>
[[end]]
```

You can pretty much use this file un-modified. It extends the current layout and embeds the `<auth/>` component into the page. It then uses `utils.app().start();` (py4web magic) to render the content of `<div id="vue">...</div>` using Vue.js. `components/auth.js` also automatically loads `components/auth.html` into the component placeholder (more py4web magic). The component is responsible for rendering the login/register/etc forms using reactive html and GETing/POSTing data to the Auth service APIs.

If you need to change the style of the component you can edit "components/auth.html" to suit your needs. It is mostly HTML with some special Vue `v-*` tags.

## 12.2 Using Auth

There two ways to use the Auth object in an action:

```
@action('index')
@action.uses(auth)
def index():
```

```
    user = auth.get_user()
    return 'hello {first_name}'.format(**user) if user else 'not logged in'
```

With `@action.uses(auth)` we tell py4web that this action needs to have information about the user, then try to parse the session for a user session.

```
@action('index')
@action.uses(auth.user)
def index():
    user = auth.get_user()
    return 'hello {first_name}'.format(**user)'
```

Here `@action.uses(auth.user)` tells py4web that this action requires a logged in user and should redirect to login if no user is logged in.

## 12.3 Auth Plugins

Plugins are defined in "py4web/utils/auth_plugins" and they have a hierachical structure. Some are exclusive and some are not. For example, default, LDAP, PAM, and SAML are exclusive (the developer has to pick one). Default, Google, Facebook, and Twitter OAuth are not exclusive (the developer can pick them all and the user gets to choose using the UI).

The `<auth/>` components will automatically adapt to display login forms as required by the installed plugins.

**At this time we cannot guarantee that the following plugins work well. They have been ported from web2py where they do work but testing is still needed**

### 12.3.1 PAM

Configuring PAM is the easiest:

```
from py4web.utils.auth_plugins.pam_plugin import PamPlugin
auth.register_plugin(PamPlugin())
```

This one like all plugins must be imported and registered. Once registered the UI (components/auth) and the RESTful APIs know how to handle it. The constructor of this plugins does not require any arguments (where other plugins do).

The `auth.register_plugin(...)` **must** come before the `auth.enable()` since it makes no sense to expose APIs before desired plugins are mounted.

### 12.3.2 LDAP

```
from py4web.utils.auth_plugins.ldap_plugin import LDAPPlugin
LDAP_SETTING = {
    'mode': 'ad',
    'server': 'my.domain.controller',
    'base_dn': 'ou=Users,dc=domain,dc=com'
}
auth.register_plugin(LDAPPlugin(**LDAP_SETTINGS))
```

### 12.3.3 OAuth2 with Google (tested OK)

```
from py4web.utils.auth_plugins.oauth2google import OAuth2Google # TESTED
```

```
auth.register_plugin(OAuth2Google(
    client_id=CLIENT_ID,
    client_secret=CLIENT_SECRET,
    callback_url='auth/plugin/oauth2google/callback'))
```

The client id and client secret must be provided by Google.

### 12.3.4 OAuth2 with Facebook (tested OK)

```
from py4web.utils.auth_plugins.oauth2facebook import OAuth2Facebook # UNTESTED
auth.register_plugin(OAuth2Facebook(
    client_id=CLIENT_ID,
    client_secret=CLIENT_SECRET,
    callback_url='auth/plugin/oauth2google/callback'))
```

The client id and client secret must be provided by Facebook.

## 12.4 Tags and Permissions

Py4web does not have the concept of groups as web2py does. Experience showed that while that mechanism is powerful it suffers from two problems: it is overkill for most apps, and it is not flexible enough for very complex apps. Py4web provides a general purpose tagging mechanism that allows the developer to tag any record of any table, check for the existence of tags, as well as checking for records containing a tag. Group membership can be thought of a type of tag that we apply to users. Permissions can also be tags. Developer are free to create their own logic on top of the tagging system.

To use the tagging system you need to create an object to tag a table:

```
groups = Tags(db.auth_user)
```

Then you can add one or more tags to records of the table as well as remove existing tags:

```
groups.add(user.id, 'manager')
groups.add(user.id, ['dancer', 'teacher'])
groups.remove(user.id, 'dancer')
```

Here the use case is group based access control where the developer first checks if a user is a member of the `'manager'` group, if the user is not a manager (or no one is logged in) py4web redirects to the `'not authorized url'`. If the user is in the correct group then py4web displays 'hello manager':

```
@action('index')
@action.uses(auth.user)
def index():
    if not 'manager' in groups.get(auth.get_user()['id']):
        redirect(URL('not_authorized'))
    return 'hello manager'
```

Here the developer queries the db for all records having the desired tag(s):

```
@action('find_by_tag/{group_name}')
@action.uses(db)
def find(group_name):
    users =
db(groups.find([group_name])).select(orderby=db.auth_user.first_name|db.auth_user.last_name)
    return {'users': users}
```

We leave it to you as an exercise to create a fixture `has_membership` to enable the following syntax:

```
@action('index')
@action.uses(has_membership(groups, 'teacher'))
def index():
    return 'hello teacher'
```

**Important:** `Tags` are automatically hierarchical. For example, if a user has a group tag 'teacher/high-school/physics', then all the following seaches will return the user:

- `groups.find('teacher/high-school/physics')`
- `groups.find('teacher/high-school')`
- `groups.find('teacher')`

This means that slashes have a special meaning for tags. Slahes at the beginning or the end of a tag are optional. All other chars are allowed on equal footing.

Notice that one table can have multiple associated `Tags` objects. The name groups here is completely arbitary but has a specific semantic meaning. Different `Tags` objects are orthogonal to each other. The limit to their use is your creativity.

For example you could create a table groups:

```
db.define_table('auth_group', Field('name'), Field('description'))
```

and to Tags:

```
groups = Tags(db.auth_user)
permissions = Tags(db.auth_groups)
```

Then create a zapper group, give it a permission, and make a user member of the group:

```
zap_id = db.auth_group.insert(name='zapper', description='can zap database')
permissions.add(zap_id, 'zap database')
groups.add(user.id, 'zapper')
```

And you can check for a user permission via an explicit join:

```
@action('zap')
@action.uses(auth.user)
def zap():
    user = auth.get_user()
    permission = 'zap database'
    if
db(permissions.find(permission))(db.auth_group.name.belongs(groups.get(user['id']))).count():
        # zap db
        return 'database zapped'
    else:
        return 'you do not belong to any group with permission to zap db'
```

Notice here `permissions.find(permission)` generates a query for all groups with the permission and we further filter those groups for those the current user is member of. We count them and if we find any, then the user has the permission.

Chapter 13

# Grid

**Warning: the API described in this chapter is new and subject to changes. Make sure you keep your code up to date**

py4web comes with a Grid object providing simple grid and CRUD capabilities.

## 13.1 Key Features

- Click column heads for sorting - click again for DESC
- Pagination control
- Search Queries list - provide the search queries used to filter the grid and grid will build the search form (remembers filters between pages)
- Filter Form - you supply and control filtering (remember filters between pages)
- Action Buttons - with or without text
- Full CRUD with Delete Confirmation
- Pre and Post Action (add your own buttons to each row)
- Grid dates in local format using moment.js
- Checkboxes in grid for boolean fields

## 13.2 Simple Example

In this simple example we will make a grid over the company table.

controllers.py

```
from py4web.utils.grid import Grid, get_storage_key, get_storage_value, GridDefaults,
 GridClassStyle
from py4web.utils.param import Param
from .settings import SESSION_SECRET_KEY

@action('administration/companies', method=['POST', 'GET'])
@action('administration/companies/<action>/<tablename>/<record_id>', method=['POST',
'GET'])
@action.uses(session, db, auth.user, requires_permission('select', 'region'),
```

```
'grid.html')
def regions(**kwargs):
    #  NOTE - GRID_COMMON would normally go in common.py and then imported to your
controller
    GRID_COMMON = GridDefaults(db=db,
                               secret=SESSION_SECRET_KEY,
                               rows_per_page=5)

    queries = [(db.company.id > 0)]
    orderby = [db.company.name]

    grid = Grid(GRID_COMMON,
                queries,
                orderby=orderby,
                storage_key=get_storage_key())
    return dict(grid=grid)
```

grid.html

```
[[extend 'layout.html']]
<div class="container" style="padding-top: 1em;">
    [[if 'action' in request.url_args and request.url_args['action'] in ['details',
'edit']:]]
        [[form = grid.render()]]
        [[=form]]
    [[else:]]
        [[=grid.render()]]
    [[pass]]
</div>
```

## 13.3 Filter/Search Example

In this simple example we will make a grid over the companies table. A search form will be created and allow searching over the company.name field.

controllers.py

```
from py4web.utils.grid import Grid, get_storage_key, get_storage_value, GridDefaults,
 GridClassStyle
from py4web.utils.param import Param
from .settings import SESSION_SECRET_KEY

@action('companies', method=['POST', 'GET'])
@action('companies/<action>/<tablename>/<record_id>', method=['POST', 'GET'])
@action.uses(session, db, auth, 'grid.html')
def companies(**kwargs):
    #  NOTE - this would normally go in common.py and then imported to your
controller
    GRID_COMMON = GridDefaults(db=db,
                               secret=SESSION_SECRET_KEY,
                               rows_per_page=5)

    search_queries = [['Search by Name', lambda value:
db.company.name.contains(values)]]

    queries = [(db.company.id > 0)]
```

```
    orderby = [db.company.name]

    grid = Grid(GRID_COMMON,
                queries,
                search_queries=search_queries,
                storage_values=dict(search_filter=search_filter),
                orderby=orderby,
                storage_key=get_storage_key())

    return dict(grid=grid)
```

The same grid.html as above is used in this example.

## 13.4 Signature

```
class Grid:
    def __init__(self,
                 common_settings,
                 queries,
                 search_form=None,
                 search_queries=None,
                 storage_values=None,
                 fields=None,
                 show_id=False,
                 orderby=None,
                 left=None,
                 headings=None,
                 create=True,
                 details=True,
                 editable=True,
                 deletable=True,
                 requires=None,
                 storage_key=None,
                 pre_action_buttons=None,
                 post_action_buttons=None):
```

- common_settings: Params object with common settings for all grids within the application
- queries: list of queries used to filter the data
- search_form: py4web FORM to be included as the search form
- search_queries: list of query lists to use to build the search form. Ignored if search_form is used. Format is
- storage_values: values to save between requests
- fields: list of fields to display on the list page, if blank, glean tablename from first query and use all fields of that table
- show_id: show the record id field on list page - default = False
- orderby: pydal orderby field or list of fields
- left: if joining other tables, specify the pydal left expression here
- headings: list of headings to be used for list page - if not provided use the field label
- create: URL to redirect to for creating records - set to False to not display the button
- editable: URL to redirect to for editing records - set to False to not display the button

- deletable: URL to redirect to for deleting records - set to False to not display the button
- requires: dict of fields and their 'requires' parm for building edit pages - dict key should be table-name.fieldname
- storage_key: id of the cookie containing saved values
- pre_action_buttons: list of action_button instances to include before the standard action buttons
- post_action_buttons: list of action_button instances to include after the standard action buttons

## 13.5 Grid Defaults

```
def __init__(self,
             db,
             secret,
             token_longevity=3600,
             rows_per_page=15,
             include_action_button_text=True,
             search_button_text="Filter",
             formstyle=FormStyleDefault,
             grid_class_style=GridClassStyle):
```

The GridDefaults class allows you to set app-wide grid defaults that you can use when instantiating grids.0

- db: PyDAL db instance to use within your grid
- secret: secret encryption key used to encrypt storage values
- token_longevity: number of seconds to remember you storage values for this grid instance
- rows_per_page: number of rows to display on each grid page
- included_action_button_text: boolean telling the grid whether or not you want text on action buttons within your grid
- search_button_text: text to appear on the submit button on your search form
- formstyle: py4web Form formstyle used to style your form when automatically building CRUD forms
- grid_class_style: GridClassStyle object used to override defaults for styling your rendered grid. Allows you to specify classes or styles to apply at certain points in the grid.

## 13.6 Searching / Filtering

There are two ways to build a search form.

- Provide a search_queries list
- Build your own custom search form

If you provide a search_queries list to grid, it will:

1. build the search form with all fields provided
2. gather filter values and filter the grid

3. if you supply a PyDAL requires statement, it will build the search field as requested

However, if this doesn't give you enough flexibility you can provide your own search form and handle all the filtering (building the queries) by yourself.PyDAL

The grid provides helper functions that allow you save/retrieve filter values between page displays.

- set_storage_values
- get_storage_value

## 13.7 CRUD

The grid provide CRUD (create, read, update and delete) capabilities utilizing py4web Form. This is disabled on the grid by default.

You can enable CRUD features by setting create/details/editable/deletable to True at instantiation.

Additionally, you can provide a separate URL to the create/details/editable/deletable parameters to bypass the auto-generated CRUD pages and handle the detail pages yourself.

## 13.8 Templates

Use the following to render your grid or CRUD forms in your templates.

Display the grid or a CRUD Form

```
[[=grid.render()]]
```

To allow for customizing CRUD form layout (like with web2py) you can use the following

```
[[form = grid.render() ]]
[[form.custom["begin"] ]]
...
[[form.custom["submit"]
[[form.custom["end"]
```

When handling custom form layouts you need to know if you are displaying the grid or a form. Use the following to decide

```
[[if 'action' in request.url_args and request.url_args['action'] in ['details',
'edit']:]]
    #  Display the custom form
    [[form = grid.render() ]]
    [[form.custom["begin"] ]]
    ...
    [[form.custom["submit"]
    [[form.custom["end"]
[[else:]]
    [[grid.render() ]]
[[pass]]
```

## 13.9 Customizing Style

You can provide your own formstyle or grid classes and style to grid.

- formstyle is the same as a Form formstyle, used to style the CRUD forms.
- grid_class_style is a class that provides the classes and/or styles used for certain portions of the grid.

The default GridClassStyle - based on no.css, primarily uses styles to modify the layout of the grid

```
def GridClassStyle(element_name):
    classes = {"wrapper": "",
               "top_div": "",
               "new_button": "",
               "search_form": "",
               "search_form_table": "search-form",
               "search_form_tr": "",
               "search_form_td": "",
               "table": "",
               "thead": "",
               "th": "",
               "sorter_icon": "",
               "action_column_header": "",
               "tbody": "",
               "tr": "",
               "td": "",
               "td_date": "",
               "td_boolean": "",
               "action_column_cell": "",
               "action_button": "",
               "table_footer": "",
               "row_count": "",
               "pager": "",
               "active_page_button": "",
               "inactive_page_button": ""}

    styles = {"wrapper": "",
              "top_div": "border-bottom: 0;",
              "new_button": "",
              "search_form": "float: right; border-bottom: 0; padding-bottom: 0;
margin-bottom: 0;",
              "search_form_table": "margin-bottom: 0;",
              "search_form_tr": "border-bottom: 0; padding-bottom: 0;",
              "search_form_td": "border-bottom: 0; padding-bottom: 0;",
              "table": "",
              "thead": "",
              "th": "text-align: center;",
              "sorter_icon": "float: right;",
              "action_column_header": "text-align: center; width: 1px; white-space:
nowrap;",
              "tbody": "",
              "tr": "",
              "td": "vertical-align: middle;",
              "td_date": "",
              "td_boolean": "",
              "action_column_cell": "text-align: center; width: 1px; white-space:
```

```
nowrap; vertical-align: middle;",
              "action_button": ("border: thin solid lightgray; "
                                "color: black; "
                                "cursor: pointer; "
                                "display: inline-block; "
                                "font-size: .75rem;"
                                "min-width: 75px; "
                                "padding-right: 1rem; "
                                "padding-left: 1rem; "
                                "text-align: center; "
                                "text-decoration: none; "
                                "vertical-align: middle; "
                                "white-space: nowrap;"),
              "table_footer": "line-height: 1.8rem; padding-bottom: 20px;",
              "row_count": "float: left; line-height: 1.8rem;",
              "pager": "float: right; line-height: 1.8rem;",
              "active_page_button": "background-color: #0074d9; "
                                    "border: thin solid #0074d9; "
                                    "color: white; "
                                    "cursor: pointer; "
                                    "display: inline-block; "
                                    "font-size: .75rem;"
                                    "padding-right: .75rem; "
                                    "padding-left: .75rem; "
                                    "margin-right: .25rem; "
                                    "text-align: center; "
                                    "text-decoration: none; "
                                    "vertical-align: middle; "
                                    "white-space: nowrap;",
              "inactive_page_button": "background-color: white; "
                                      "border: thin solid #0074d9; "
                                      "color: #0074d9; "
                                      "cursor: pointer; "
                                      "display: inline-block; "
                                      "font-size: .75rem;"
                                      "padding-right: .75rem; "
                                      "padding-left: .75rem; "
                                      "margin-right: .25rem; "
                                      "text-align: center; "
                                      "text-decoration: none; "
                                      "vertical-align: middle; "
                                      "white-space: nowrap;"}

    classes_styles = {}
    if classes.get(element_name) and classes.get(element_name) != "":
        classes_styles["_class"] = classes.get(element_name)

    if styles.get(element_name) and styles.get(element_name) != "":
        classes_styles["_style"] = styles.get(element_name)

    return classes_styles
```

GridClassStyleBulma - bulma implementation

```
def GridClassStyleBulma(element_name):
    classes = {"wrapper": "field",
               "top_div": "pb-2",
               "new_button": "button",
               "search_form": "is-pulled-right pb-2",
```

```
                "search_form_table": "search-form",
                "search_form_tr": "",
                "search_form_td": "pr-1",
                "table": "table is-bordered is-striped is-hoverable is-fullwidth",
                "thead": "",
                "th": "has-text-centered",
                "sorter_icon": "is-pulled-right",
                "action_column_header": "has-text-centered is-narrow",
                "tbody": "",
                "tr": "",
                "td": "",
                "td_date": "has-text-centered",
                "td_boolean": "has-text-centered",
                "action_column_cell": "has-text-centered is-narrow",
                "action_button": "button is-small",
                "table_footer": "",
                "row_count": "is-pulled-left",
                "pager": "is-pulled-right",
                "active_page_button": "button is-primary is-small",
                "inactive_page_button": "button is-small"}

    styles = {"wrapper": "",
              "top_div": "",
              "new_button": "",
              "search_form": "",
              "search_form_table": "",
              "search_form_tr": "",
              "search_form_td": "",
              "table": "",
              "thead": "",
              "th": "",
              "sorter_icon": "",
              "action_column_header": "",
              "tbody": "",
              "tr": "",
              "td": "",
              "td_date": "",
              "td_boolean": "",
              "action_column_cell": "",
              "action_button": "",
              "table_footer": "",
              "row_count": "",
              "pager": "",
              "active_page_button": "",
              "inactive_page_button": ""}

    classes_styles = {}
    if classes.get(element_name) and classes.get(element_name) != "":
        classes_styles["_class"] = classes.get(element_name)

    if styles.get(element_name) and styles.get(element_name) != "":
        classes_styles["_style"] = styles.get(element_name)

    return classes_styles
```

## 13.10 Custom Action Buttons

As with web2py, you can add additional buttons to each row in your grid. You do this by providing pre_action_buttons or post_action_buttons to the Grid **init** method.

- pre_action_buttons - list of action_button instances to include before the standard action buttons
- post_action_buttons - list of action_button instances to include after the standard action buttons

## 13.11 Action Button Signature

```
def __init__(self,
             url,
             text,
             icon="fa-calendar",
             additional_classes=None,
             additional_styles=None,
             override_classes=None,
             override_styles=None,
             message=None,
             append_id=False,
             append_storage_key=False,
             append_page=False):
```

- url: the page to navigate to when the button is clicked
- text: text to display on the button
- icon: the font-awesome icon to display before the text
- additional_classes: a space-separated list of classes to include on the button element
- additional_styles: a space-separated list of classes to include on the button element
- override_classes: a space-separated list of classes to override the defaults set up for a specific button
- override_styles: a space-separated list of classes to override the defaults set up for a specific button
- message: confirmation message to display if 'confirmation' class is added to additional classes
- append_id: if True, add id_field_name=id_value to the url querystring for the button
- append_storage_key: if True, append the storage key for the grid to the url for the button
- append_page: if True, append page=`page_number` to the url querystring

Grid uses ActionButtons internally to generate the row buttons in the grid. You can provide your own by specifying a list of ActionButtons in the pre_action_buttons and/or post_action_buttons parameter on the Grid **init** method.

## 13.12 Reference Fields

When displaying fields in a PyDAL table, you sometimes want to display a more descriptive field than a foreign key value. There are a couple of ways to handle that with the py4web grid.

filter_out on PyDAL field definition - here is an example of a foreign key field

```
Field('company', 'reference company',
      requires=IS_NULL_OR(IS_IN_DB(db, 'company.id',
                                   '%(name)s',
                                   zero='..')),
      filter_out=lambda x: x.name if x else ''),
```

This will display the company name in the grid instead of the company ID

The downfall of using this method is that sorting and filtering are based on the company field in the employee table and not the name of the company

left join and specify fields from joined table - specified on the left parameter of Grid instantiation

```
db.company.on(db.employee.company == db.company.id)
```

You can specify a standard PyDAL left join, including a list of joins to consider.

Now the company name field can be included in your fields list can be clicked on and sorted.

Now you can also specify a query such as:

```
queries.append((db.employee.last_name.contains(search_text)) |
(db.employee.first_name.contains(search_text)) |
db.company.name.contains(search_text)))
```

This method allows you to sort and filter, but doesn't allow you to combine fields to be displayed together as the filter_out method would

You need to determine which method is best for your use case understanding the different grids in the same application may need to behave differently.

- *Index*
- *Module Index*
- *Search Page*