py4web Documentation

Release 1.2025-preview

Massimo Di Pierro

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 to 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. 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, since 2015 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. 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 (see Chapter 15 for more details if you come from web2py):

- PY4WEB, unlike web2py, requires Python 3.
- PY4WEB, unlike web2py, can be installed using pip, and its dependencies are managed using pyproject.toml
- PY4WEB apps are regular Python modules. This is very different from 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 ombott (a reduced and faster spin-off of Bottle) and in particular uses a Bottle-compatible request object and 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 arrives or needs cleanup when completed. This makes it much faster than web2py.

- PY4WEB, has a new session 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 optionally 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 leverages 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 Form and Grid objects that are similar to web2py's SQLFORM and SQLFORM.grid.

- 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 leverages PyDAL's new tags functionality to tag users with groups, 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.

1.1 Acknowledgments

Many thanks to everyone who has contributed to the project, and especially:

Special thanks to Sam de Alfaro, who designed the official logo of py4web. We friendly call the logo "Axel the axolotl": it magically represents the sense of kindness and inclusion. We believe it's the cornerstone of our growing community.

Help, resources and hints

We've made our best to make PY4WEB simple and clean. But you know, modern web programming is a daunting task. It requires an open mind, able to jump frequently (without being lost!) from python to HTML to javascript to css and even database management. But don't be scared, in this manual we'll assist you side by side in this journey. And there are many other valuable resources that we're going to show you.

2.1 Resources

2.1.1 This manual

This manual is the Reference Manual for py4web. It's available online at https://py4web.com/_documentation/static/index.html, where you'll also find the PDF and EBOOK version, in multiple languages. It written in RestructuredText and generated using Sphinx.

2.1.2 The Google group

There is a dedicated mailing list hosted on Google Groups, see https://groups.google.com/g/py4web. This is the main source of discussions for developers and simple users. For any problem you should face, this is the right place to search for a hint or a solution.

2.1.3 The Discord server

For quick questions and chats you can also use the free Discord server dedicated to py4web. You could usually find many py4web developers hanging out in the channel.

2.1.4 Tutorials and video

There are many tutorials and videos available. Here are some of them:

- the Learn Py4Web site by Luca de Alfaro (with lots of excellent training videos)

- the free video course 2020 by Luca de Alfaro at UC Santa Cruz

- the py4web blog app by Andrew Gavgavian, which uses py4web to replicate the famous Corey Schafer's tutorial series on creating a blog app in Django

- the South Breeze Enterprises demo app by Jim Steil. It is built around the structure of the Microsoft Northwind database, but converted to SQLite. You can view the final result online here

2.1.5 The sources on GitHub

Last but not least, py4web is Open Source, with a BSD v3 license, hosted on GitHub at https://github.com/web2py/py4web. This means that you can read, study and experiment with all of its internal details by yourself.

2.2 Hints and tips

This paragraph is dedicated to preliminary hints, suggestions and tips that could be helpful to know before starting to learn py4web.

2.2.1 Prerequisites

In order to understand py4web you need at least a basic python knowledge. There are many books, courses and tutorials available on the web - choose what's best for you. The python's decorators, in particular, are a milestone of any python web framework and you have to fully understand it.

2.2.2 A modern python workplace

In the following chapters you will start coding on your computer. We suggest you to setup a modern python workplace if you plan to do it efficiently and safely. Even for running simple examples and experimenting a little, we strongly suggest to use an **Integrated Development Environment** (IDE). This will make your programming experience much better, allowing syntax checking, linting and visual debugging. Nowadays there are two free and multi-platform main choices: Microsoft Visual Studio Code aka VScode and JetBrains PyCharm.

When you'll start to deal with more complex programs and need reliability, we also suggest to:

- use virtual environments (also called **virtualenv**, see here for an introduction). In a complex workplace this will avoid to be messed up with other python programs and modules

- use **git** to keep track of your program's changes and save your changes in a safe place online (GitHub, GitLat, or Bitbucket).

- use an editor with Syntax Highlighting. We highly recommend Visual Studio Code (VScode) or PyCharm.

2.2.3 Debugging py4web with VScode

It's quite simple to run and debug py4web within VScode.

If you have **installed py4web from source**, you just need to open the main py4web folder (not the apps folder!) with VScode and add:

```
"args": ["run", "apps"],
"program": "your_full_path_to_py4web.py",
```

to the vscode `launch.json` configuration file. Note that if you're using Windows the "your_full_-path_to_py4web.py" parameter must be written using forward slash only, like "C:/Users/your_-name/py4web/py4web.py".

If you have instead **installed py4web from pip,** you need to set the launch.json file to run py4web as a module

```
{
  "name": "py4web apps",
  "type": "debugpy",
  "request": "launch",
  "module": "py4web",
  "args": ["run", "apps", "-D", "--watch", "lazy"]
}
```

Adjust the *args* to match your apps folder. For example, replace *apps* with . if you opened the apps folder itself in VSCode.

Tip In both cases, if you should get gevent errors you have to also add `"gevent": true` on

---

the `launch.json` configuration file.

2.2.4 Debugging py4web with PyCharm

In PyCharm, if you should get gevent errors you need to enable Settings | Build, Execution, Deployment | Python Debugger | Gevent compatible.

2.3 How to contribute

We need help from everyone: support our efforts! You can just participate in the Google group trying to answer other's questions, submit bugs using or create pull requests on the GitHub repository.

If you wish to correct and expand this manual, or even translate it in a new foreign language, you can read all the needed information directly on the specific README on GitHub.

It's really simple! Just change the .RST files in the /doc folder and create a Pull Request on the GitHub repository at https://github.com/web2py/py4web - you can even do it within your browser. Once the PR is accepted, your changes will be written on the master branch, and will be reflected on the web pages / pdf / epub at the next output generation on the branch.

Installation and Startup

3.1 Understanding the design

Before everything else it is important to understand that unlike other web frameworks, is not only a python module that can be imported by apps. It is also a program that is in charge of starting some apps. For this reason you need two things:

- The py4web module (which you download from our web site, from pypi or from github)
- One or more folders containing collections of apps you want to run.

py4web has command line options to create a folder with some example apps, to initialize an existing folder, and to add scaffolding apps to that folder. Once installed you can have multiple apps under the same folder running concurrently and served by the same py4web process at the same address and port. An apps folder is a python module, and each app is also a python module.

3.2 Supported platforms and prerequisites

py4web runs fine on Windows, MacOS and Linux. Its only prerequisite is Python 3.7+, which must be installed in advance (except if you use binaries).

3.3 Setup procedures

There are four alternative ways of installing py4web, we will guide you through each of them and if you get stuck, reach out to us.

3.3.1 Installing from pip, using a virtual environment

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!

Here are the instructions for creating the virtual environment, activating it, and installing py4web in it:

Linux and MacOS

```
python3 -m venv venv
. venv/bin/activate
python -m pip install --upgrade py4web --no-cache-dir
```

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

Starting py4web is same with or without a virtual environment python py4web run apps

Windows

```
run cmd.exe
In e.g. folder c:\py4web
python3 -m venv venv
"C:\py4web\venv\Scripts\activate.bat"
python -m pip install --upgrade py4web --no-cache-dir
cd venv\scripts
py4web.exe setup apps
py4web.exe set_password
py4web.exe run apps
```

You can also find power shell scripts in the same folder. Starting py4web is same with or without a virtual environment python py4web run apps

3.3.2 Installing from pip, without virtual environment

pip is the basic installation procedure for py4web, it will quickly install the latest stable release of py4web.

From the command line

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

Also, if `python3` does not work, try specify a full version as in `python3.8`.

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, then you can run the needed commands like this

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

3.3.3 Installing from source (globally)

This is the traditional way for installing a program, but it works only on Linux and MacOS (Windows does not normally support the *make* utility). 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
python -m pip install .
py4web setup apps
py4web set_password
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.

Notice that you also (and should) install py4web from source inside a virtual environment.

3.3.4 Running from source without installing

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. Also, installing from sources (locally or globally) will install all the latest changes present on the master branch of py4web - hence you will gain the latest (but potentially untested) code.

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 -e .
```

Once installed, you should always start it from there with:

Linux and 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.

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 **./** as on Linux). But running .py files directly it's not usual and you'll need an explicit python3/python command.

3.3.5 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 beginners 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, DAL support is limited 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 set_password
./py4web run apps
```

(omit './' if you're using Windows).

Notice: the binaries many not correspond to the latest master or the latest stable branch of py4web although we do our best to keep them up to date.

3.4 Upgrading

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

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

> **Warning** This will not automatically upgrade the standard apps like **Dashboard** and **Default**. You have to manually remove these apps and then run
>
> ```
> py4web setup <path to apps_folder>
> ```
>
> 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.

3.4.1 Running Using uv
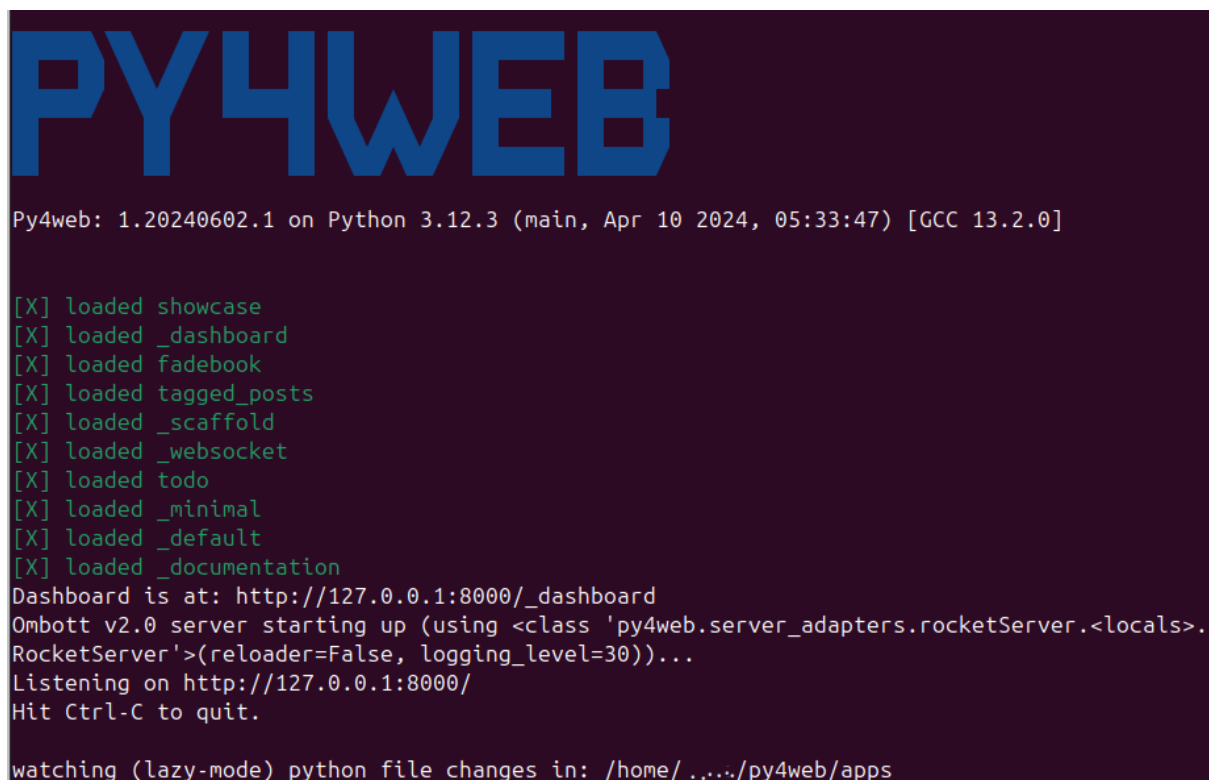
This is the newest way to manage python packages. Install uv as shown here: https://docs.astral.sh/uv/getting-started/installation/ Then run:

```
uv run py4web.py run apps
```

More uv command examples are in the provided Makefile

3.5 First run

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

```
PY4WEB

Py4web: 1.20240602.1 on Python 3.12.3 (main, Apr 10 2024, 05:33:47) [GCC 13.2.0]

[X] loaded showcase
[X] loaded _dashboard
[X] loaded fadebook
[X] loaded tagged_posts
[X] loaded _scaffold
[X] loaded _websocket
[X] loaded todo
[X] loaded _minimal
[X] loaded _default
[X] loaded _documentation
Dashboard is at: http://127.0.0.1:8000/_dashboard
Ombott v2.0 server starting up (using <class 'py4web.server_adapters.rocketServer.<locals>.
RocketServer'>(reloader=False, logging_level=30))...
Listening on http://127.0.0.1:8000/
Hit Ctrl-C to quit.

watching (lazy-mode) python file changes in: /home/...../py4web/apps
```

Generally `apps` is the name of the folder where you keep all your apps, and can be explicitly set wit the `run` command. (Yet nothing prevents you from grouping apps in multiple folders with different names.) 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. It will be described in the next chapter.

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

Note 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 from the local machine:

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

In order to stop py4web, you need to hit `Control-C` on the window where you run it.

Note 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 may want to symlink `apps/_default` to your default app.

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

> **Warning** For Windows: it could be that `Ctrl-C` does not work in order to stop py4web. In this case, try with `Ctrl-Break` or `Ctrl-Fn-Pause`.

3.6 Command line options

py4web provides multiple command line options which can be listed by running it without any argument

```
# py4web
```

```
Usage: py4web.py [OPTIONS] COMMAND [ARGS]...

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

  Type "./py4web.py COMMAND -h" for available options on commands

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

Commands:
  call          Call a function inside apps_folder
  new_app       Create a new app copying the scaffolding one
  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
```

You can have additional help for a specific command line option by running it with the **–help** or **-h** argument.

3.6.1 `call` command option

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

```
  Call a function inside apps_folder

Options:
  -Y, --yes          No prompt, assume yes to questions  [default: False]
  --args TEXT        Arguments passed to the program/function  [default: {}]
  -help, -h, --help  Show this message and exit.
```

For example:

```
# py4web call apps examples.test.myfunction --args '{"x": 100}'
```

where myfunction is the function you want to call in apps/examples/test.py. Note that you have to use the single and double quotes just as shown for parameters to work.

3.6.2 `new_app` command option

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

  Create a new app copying the scaffolding one

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

  -s, --scaffold_zip TEXT  Path to the zip with the scaffolding app
  -help, -h, --help        Show this message and exit.
```

This currently gives an error on binaries installations and from source installation (locally), because they miss the asset zip file.

3.6.3 `run` command option

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

  Run the applications on apps_folder

Options:
  -Y, --yes                    No prompt, assume yes to questions
  -H, --host TEXT              Host listening IP  [default: 127.0.0.1]
  -P, --port INTEGER           Port number  [default: 8000]
  -A, --app_names TEXT         List of apps to run, comma separated (all if
                                 omitted or empty)
  -p, --password_file TEXT     File for the encrypted password  [default:
                                 password.txt]
  -Q, --quiet                  Suppress server output
  -R, --routes                 Write apps routes to file
  -s, --server
[default|wsgiref|tornado|wsgiref+threaded|rocket|waitress|gunicorn|gevent|gunicorn+gevent|gevent
                                 Web server to use (unavailable: waitress,
                                 gunicorn, gevent, gunicorn+gevent,
                                 gevent+websockets)
  -w, --number_workers INTEGER Number of workers  [default: 0]
  -d, --dashboard_mode TEXT    Dashboard mode: demo, readonly, full, none
                                 [default: full]
  --watch [off|sync|lazy]      Watch python changes and reload apps
                                 automatically, modes: off, sync, lazy
                                 [default: lazy]
  --ssl_cert PATH              SSL certificate file for HTTPS
  --ssl_key PATH               SSL key file for HTTPS
  --errorlog TEXT              Where to send error logs
                                 (:stdout|:stderr|tickets_only|{filename})
```

```
                                             [default: :stderr]
  -L, --logging_level INTEGER     The log level (0 - 50) [default: 30
                                     (=WARNING)]
  -D, --debug                     Debug switch
  -U, --url_prefix TEXT           Prefix to add to all URLs in and out
  -m, --mode TEXT                 default or development  [default: default]
  -help, -h, --help               Show this message and exit.
```

The `app_names` option lets you filter which specific apps you want to serve (comma separated). If absent or empty all the apps in the APPS_FOLDER will be run.

By default (for security reasons) the py4web framework will listen only on 127.0.0.1, i.e. localhost. If you need to reach it from other machines you must specify the host option, like `py4web run --host 0.0.0.0 apps`.

The `url_prefix` option is useful for routing at the py4web level. It allows mapping to multiple versions of py4web running on different ports as long as the url_prefix and port match the location. For example `py4web run --url_prefix=/abracadabra --port 8000 apps`.

By default py4web will automatically reload an application upon any changes to the python files of that application. The reloading will occur on any first incoming request to the application that has been changed (lazy-mode). If you prefer an immediate reloading (sync-mode), use `py4web run --watch sync`. For production servers, it's better to use `py4web run --watch off` in order to avoid unneded checks (but you will need to restart py4web for activating any change).

Note The `--watch` directive looks for any changes occurring to the python files under the `/apps` folder only. Any modifications to the standard py4web programs will always require a full restart of the framework.

The default web server used is currently rocketServer, but you can change this behaviour with the `server` option. Rocket3 is the multi-threaded web server used by web2py stripped of all the Python2 logic and dependencies.

The logging_level values are defined in the **logging** standard python module. The default value is 30 (it corresponds to WARNING). Other common values are 0 (NOTSET), 10 (DEBUG), 20 (INFO), 40 (ERROR) and 50 (CRITICAL). Using them, you're telling the library you want to handle all events from that level on up.

The debug parameter automatically sets logging_level to 0 and logs all calls to fixture functions. It also logs when a session is found, invalid, saved.

3.6.4 `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

```
py4web 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: *****
```

3.6.5 `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.

3.6.6 `shell` command option

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

  Open a python shell with apps_folder's parent added to the path

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

3.6.7 `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 -all option you'll get the version of all the available python modules, too.

3.7 Special installations

There are special cases in which you cannot or don't want to use one of the generic installation instructions we've already described. There is a special folder called `deployment_tools` in the py4web repository that collects some special recipes. They are briefly described here, along with some tips and tricks.

3.7.1 HTTPS

To use https with the build-in web server (Rocket3) these are the steps:

- Generate the localhost certificates. For example followed the instructions here:

    https://www.section.io/engineering-education/how-to-get-ssl-https-for-localhost/.

- Restart your browser and browse securely to your web site.

If you use VSCode to run py4web you may want to update the py4web launch.json file to contain:

```
"configurations": [
        {
            "name": "py4web",
            "type": "debugpy",
            "request": "launch",
            "module": "py4web",
            // or "program": "${workspaceFolder}/py4web.py", if you didn't install
py4web as a package
            "args": [
                "run",
                "apps",
                "--ssl_cert", "/path_to/localhost.crt",
                "--ssl_key", "/path_to/localhost.key",
                "--server", "rocketServer",
            ]
        }
    ]
```

Notice that /path_to/ should be the absolute path to the location of your certificate.

3.7.2 WSGI

py4web is a standard WSGI application. So, if a full program installation it's not feasible you can simply run py4web as a WSGI app. For example, using gunicorn-cli, create a python file:

```
# py4web_wsgi.py
from py4web.core import wsgi
application = wsgi(apps_folder="apps")
```

and then start the application using cli:

```
gunicorn -w 4 py4web_wsgi:application
```

The wsgi function takes arguments with the same name as the command line arguments.

3.7.3 Deployment on GCloud (aka GAE - Google App Engine)

Login into the Gcloud console 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`.

3.7.4 Deployment on PythonAnywhere.com

Watch the YouTube video and follow the detailed tutorial . The bottle_app.py script is in `py4web/deployment_tools/pythonanywhere.com/bottle_app.py`

3.7.5 Deployment on Docker/Podman

On `deployment_tools/docker` there is a simple Dockerfile for quickly running a py4web container. There is also a docker-compose.yml file for setting up a more complex multi-container with PostgreSQL. A ready docker example based on the Scaffold application can be cloned from this repository <https://github.com/macneiln/docker-py4web-scaffold>

Note that you can use them also with Podman, which has the advantage of does not requiring sudo and does not running any background daemon.

3.7.6 Deployment on Ubuntu

On `deployment_tools/ubuntu` there is a bash script tested with Ubuntu Server 20.04.03 LTS. It uses nginx and self-signed certificates. It optionally manage iptables, too.

The Dashboard

The Dashboard is the standard web based IDE; you will surely use it extensively to manage the applications and check your databases. Looking at its interface is a good way to start exploring py4web and its components.

4.1 The main Web page

When you run the standard py4web program, it starts a web server with a main web page listening on http://127.0.0.1:8000 (which means that it is listening on the TCP port 8000 on your local PC, using the HTTP protocol).

You can connect to this main page only from your local PC, using a web browser like Firefox or Google Chrome:
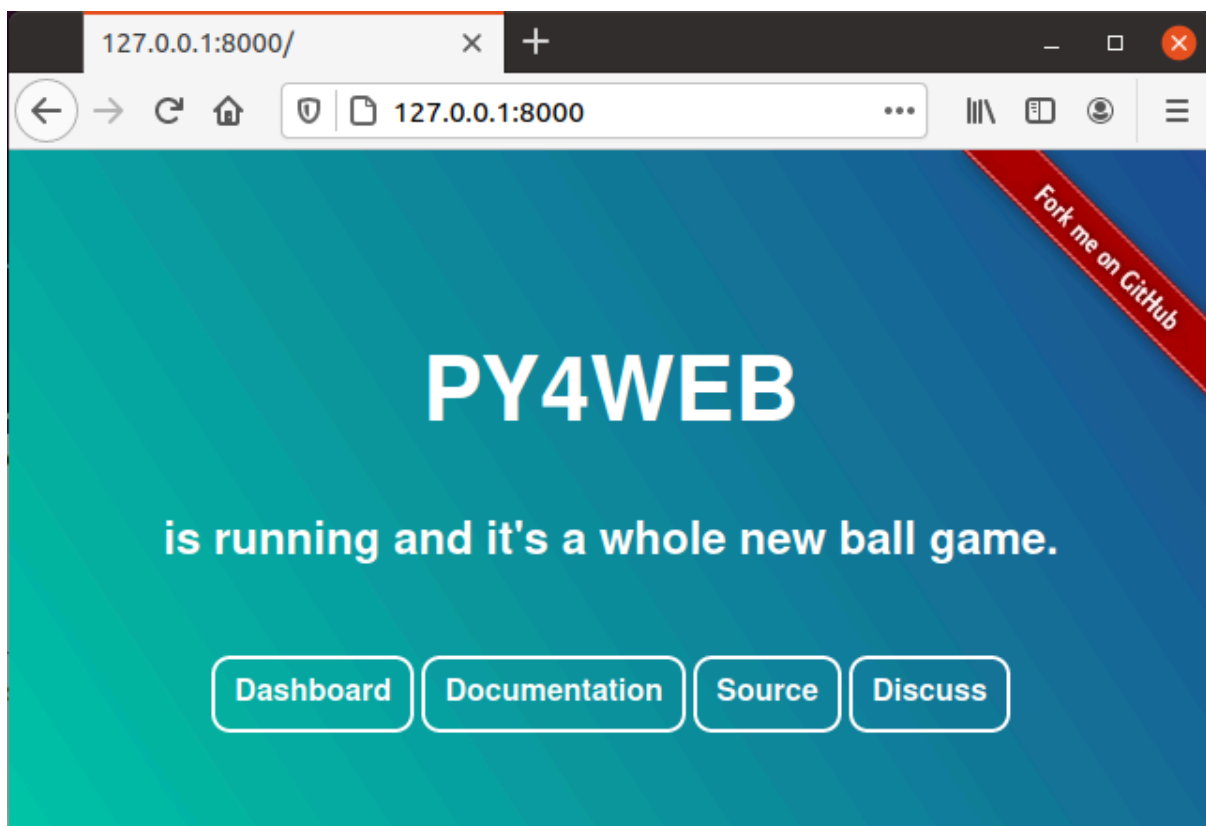
The buttons are:

- Dashboard (http://127.0.0.1:8000/_dashboard), which we'll describe in this chapter.

- Documentation (http://127.0.0.1:8000/_documentation?version=1.20201112.1), for browsing the local copy of this Manual.
- Source (https://github.com/web2py/py4web), pointing to the GitHub repository.
- Discuss (https://groups.google.com/forum/#!forum/py4web), pointing to the Google mail group.

4.2 Login into the Dashboard

Pressing the Dashboard button will forward you to the Dashboard login. Here you must insert the password that you've already setup (see Section 3.6.4). If you don't remember the password, you have to stop the program with CTRL-C, setup a new one and run the py4web again.
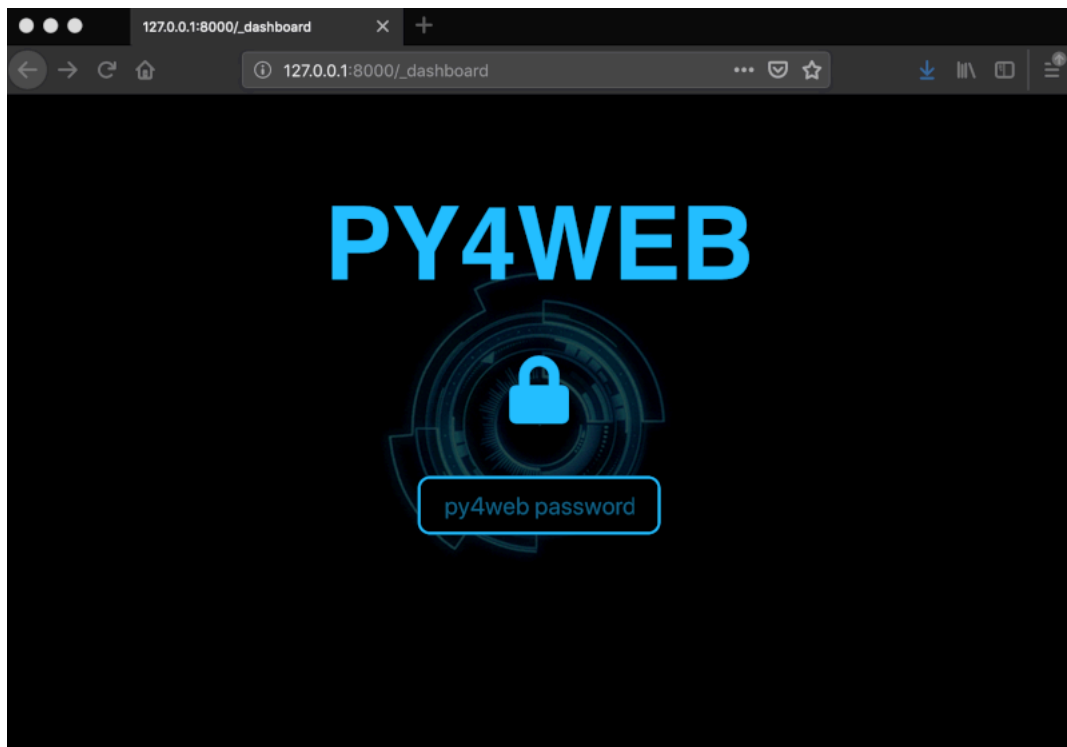
After inserting the right Dashboard's password, it will be displayed with all the tabs compressed.
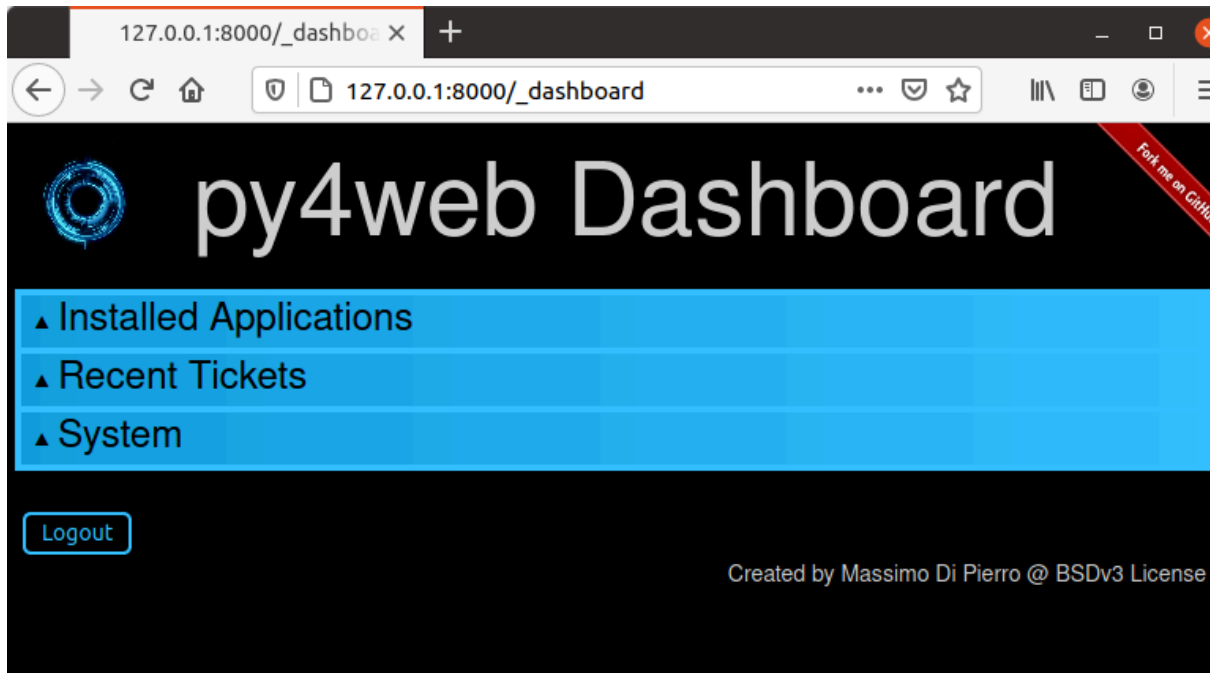
Click on a tab title to expand. Tabs are context dependent. For example, open tab "Installed Applications" and click on an installed application to select it.

This will create new tabs "Routes", "Files", and "Model" for the selected app.
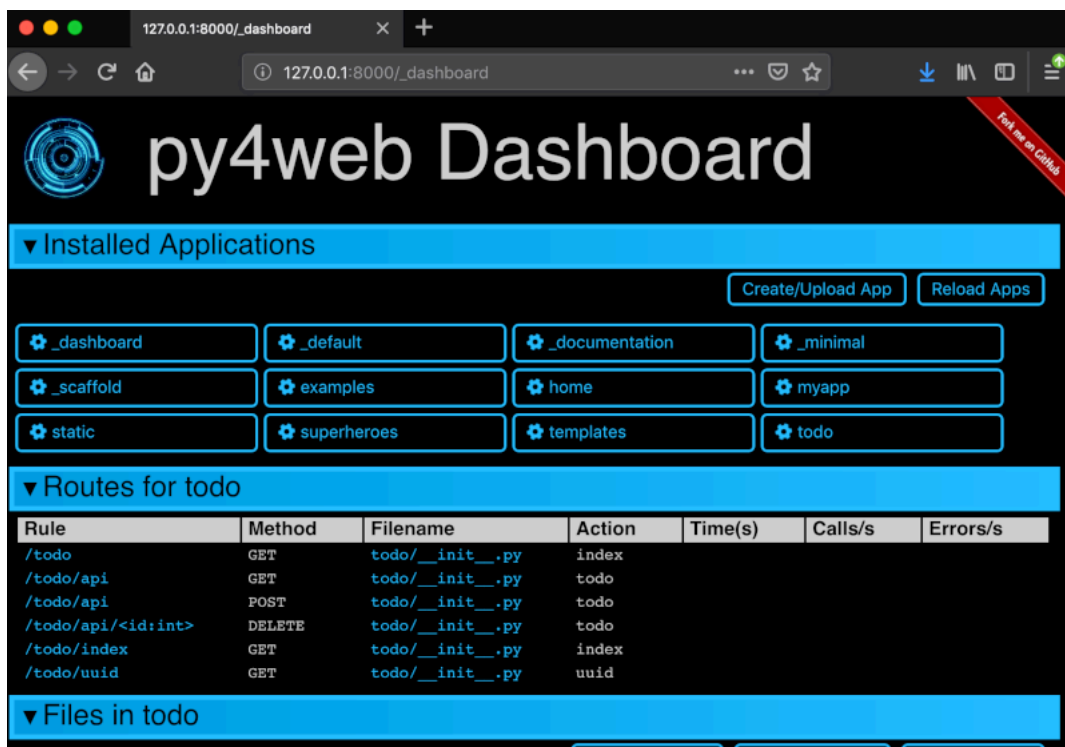
The "Files" tab allows you to browse the folder that contains the selected app and edit any file that comprises the app. If you edit a file by default it will be automatically reloaded at its first usage (unless you've changed the *watch* option with the Section 3.6.3; in this case you must click on "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 an app and triggers a bug, the user is issued a ticket.
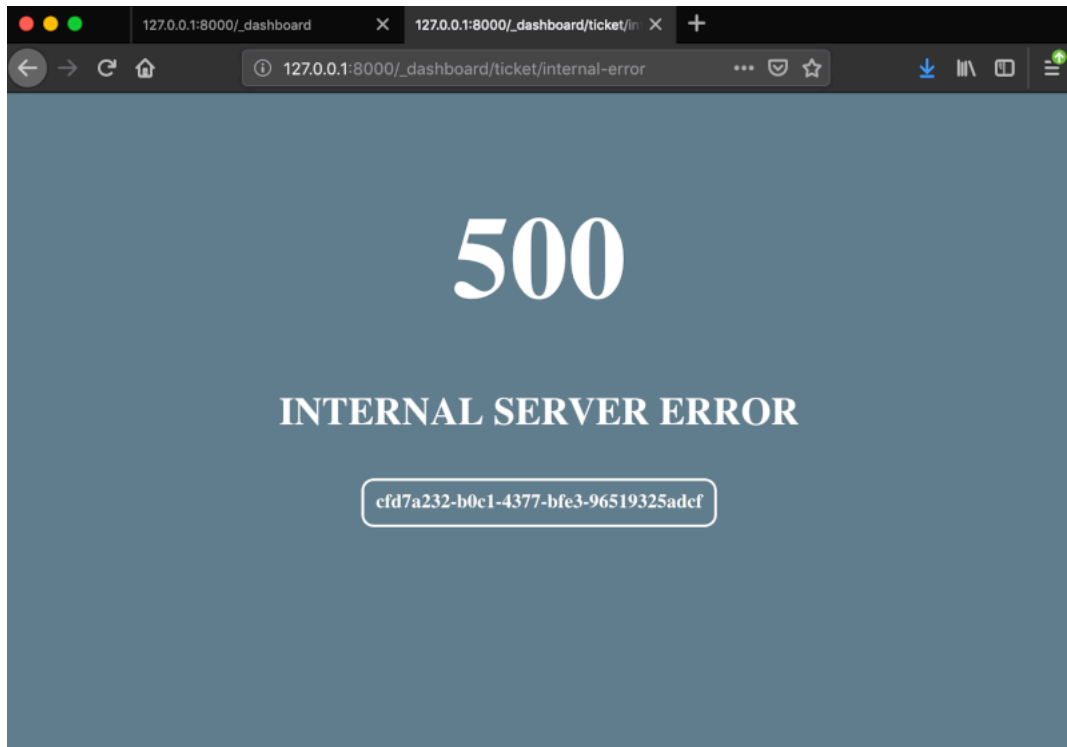
The ticket is logged in py4web database. The Dashboard displays the most common recent issues and allows searching tickets.
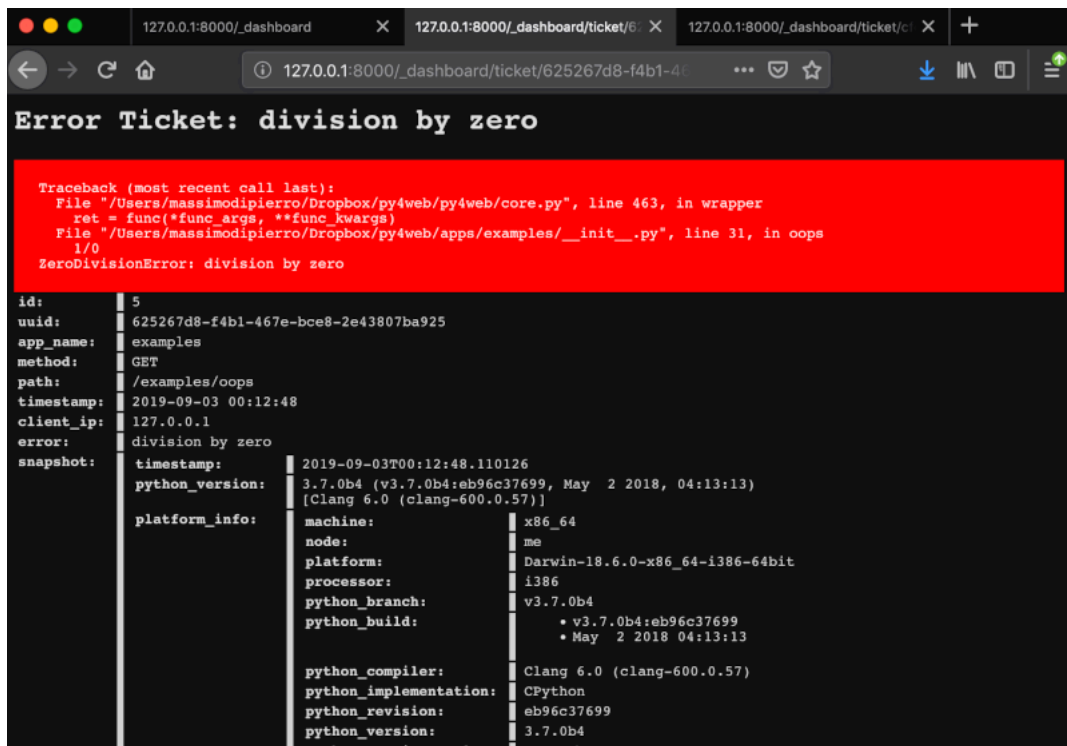
Creating an app

5.1 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 already 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.

Note An empty *__init__.py* file is not strictly needed since Python 3.3, but it will be useful later on.

Open a command prompt and go to your main py4web folder. Enter the following simple commands in order to create a new empty **myapp** app:

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

Tip for Windows, you must use backslashes (i.e. \) instead of slashes.

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. By default py4web runs in *lazy watch* mode (see the Section 3.6.3) for automatic reloading of the apps whenever it changes, which is very useful in a development environment. In production or debugging environment, it's better to run py4web with a command like this:

```
py4web run apps --watch off
```

A py4web 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.

5.2 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 ombott (One More BOTTle) web server, which is a minimal and fast bottlepy spin-off. It supports streaming, partial content, range requests, and if-modified-since. This is all handled automatically based on the HTTP request headers.

5.3 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:

```python
import datetime
from py4web import action

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

Reload the app, 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.

5.3.1 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 at the end

```python
@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.

5.3.2 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
- `<name:filter:config>`

And these are possible filters (only `:re` 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`.

Note that the routing is implemented in ombott as radix-tree hybrid router. It is declaration-order-independent and it prioritizes static route-fragment over dynamic one, since this is most expected behavior.

This results in some constraints, such as one cannot have more than one route that has dynamic fragment of different types (int, path) in the same place.. Hence **something like this is incorrect** and will result in errors:

```
@action('color/<code:int>')
def color(code):
    return f'Color code: {code}'


@action('color/<name:path>')
def color(name):
    return f'Color name: {name}'
```

Instead, to accomplish a simmilar result, one needs to handle all the logic in one action:

```
@action('color/<color_identifier:path>')
def color(color_identifier):
    try:
        msg = f'Color code: {int(color_identifier)}'
    except:
        msg = f'Color name: {color_identifier}'
    return msg
```

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.

5.3.3 The `request` object

From py4web you can import `request`

```
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 equivalent to a Bottle request object. with one additional attribute:

```
request.app_name
```

Which you can use the code to identify the name and the folder used for the app.

5.3.4 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>
 <head>
    <style>
      body {background:[[=color]]}
    </style>
 </head>
 <body>
    <h1>Color [[=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.

5.4 The _scaffold app

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. Also, it shows you how to manage users and their registration. Just like a real scaffolding in a building construction site, scaffolding could give you some kind of a fast and simplified structure for your project, on which you can rely to build your real project.
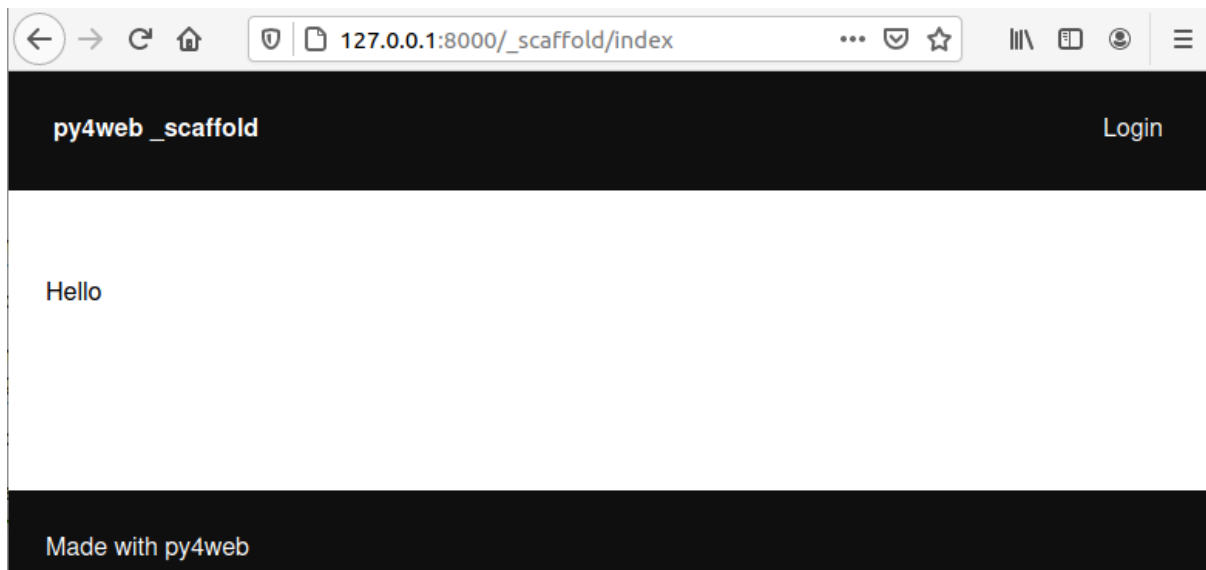
You will normally find the scaffold app under apps, but you can easily create a new clone of it manually or using the Dashboard.

Here is the tree structure of the _scaffold app:

```
_scaffold/
├── common.py
├── controllers.py
├── __init__.py
├── models.py
├── settings.py
├── static
│   ├── css
│   │   └── no.css
│   ├── favicon.ico
│   ├── js
│   │   └── utils.js
│   └── README.md
├── tasks.py
└── templates
    ├── auth.html
    ├── generic.html
    ├── index.html
    ├── layout.html
    └── README.md
```
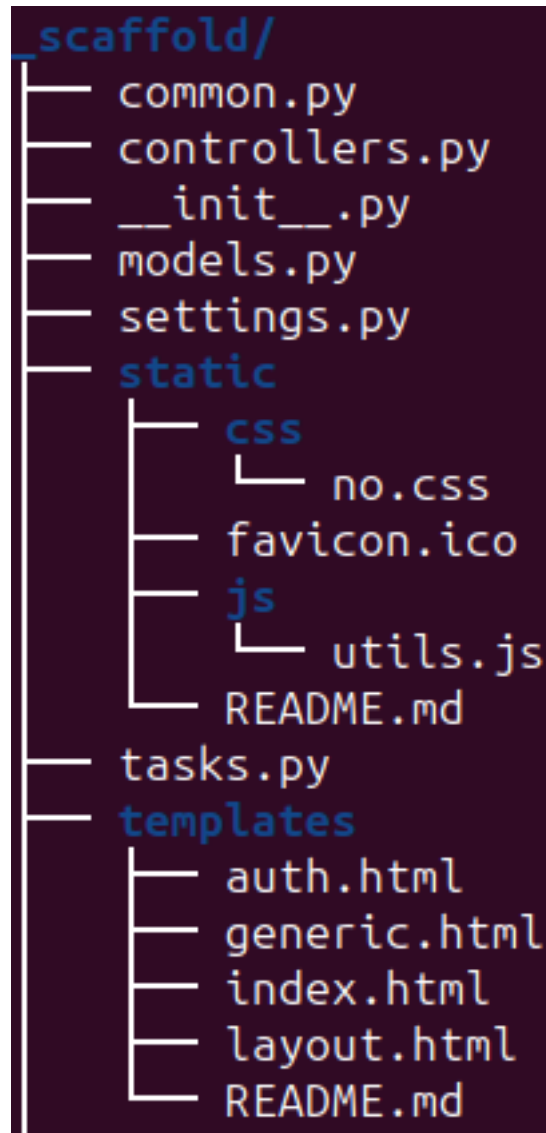
The scaffold app contains an example of a more complex action:

```python
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 `ombott`.

- `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.

5.5 Copying the _scaffold app

The scaffold app is really useful, and you will surely use it a lot as a starting point for testing and even developing full features new apps.

It's better not to work directly on it: always create new apps copying it. You can do it in two ways:

- using the command line: copy the whole apps/_scaffold folder to another one (apps/my_app for example). Then reload py4web and it will be automatically loaded.
- using the Dashboard: select the button `Create/Upload App` under the "Installed Applications" upper section. Just give the new app a name and check that "Scaffold" is selected as the source. Finally press the `Create` button and the dashboard will be automatically reloaded, along with the new app.
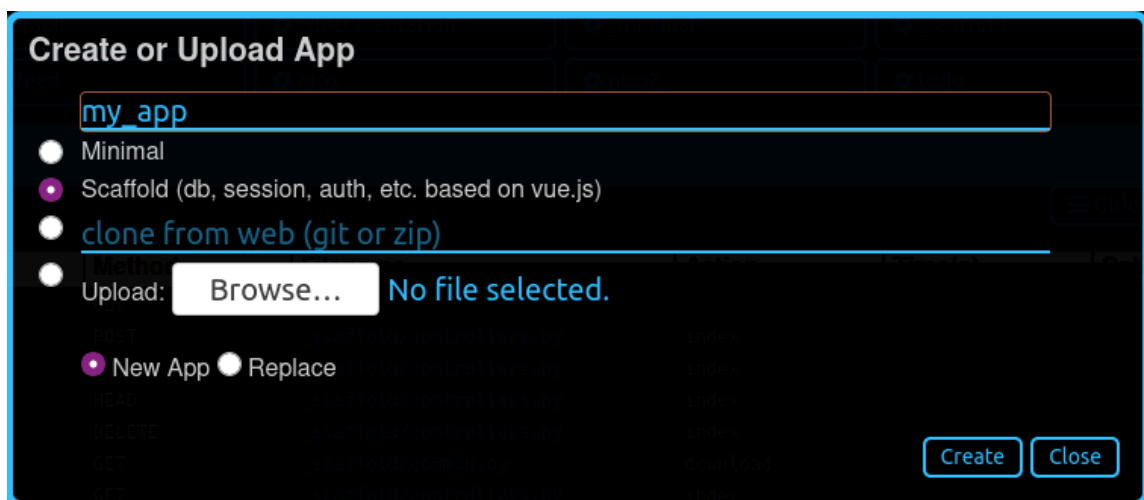
5.6 Watch for files change

As described in the Section 3.6.3, Py4web facilitates a development server's setup by automatically reloads an app when its Python source files change (by default). But in fact any other files inside an app can be watched by setting a handler function using the `@app_watch_handler` decorator.

Two examples of this usage are reported now. Do not worry if you don't fully understand 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
```

```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:

```python
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 (i.e.
"*.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.

5.7 Domain-mapped apps

In production environments it is often required to have several apps being served by a single py4web
server, where different apps are mapped to different domains.

py4web can easily handle running multiple apps, but there is no build-in mechanism for mapping
domains to specific applications. Such mapping needs to be done externally to py4web – for instance
using a web reverse-proxy, such as nginx.

While nginx or other reverse-proxies are also useful in production environments for handling SSL
termination, caching and other uses, we cover only the mapping of domains to py4web applications
here.

An example nginx configuration for an application `myapp` mapped to a domain
`myapp.example.com` might look like that:

```
server {
   listen 80;
   server_name myapp.example.com;
   proxy_http_version 1.1;
   proxy_set_header Host $host;
   proxy_set_header X-PY4WEB-APPNAME /myapp;
   location / {
      proxy_pass http://127.0.0.1:8000/myapp$request_uri;
   }
}
```

This is an example `server` block of nginx configuration. One would have to create a separate such
block for **each app/each domain** being served by py4web server. Note some important aspects:

- `server_name` defines the domain mapped to the app `myapp`,

- **proxy_http_version 1.1; directive is optional, but highly recommended (otherwise nginx uses HTTP 1.0 to talk**

    to the backend-server – here py4web – and it creates all kinds of issues with buffering and otherwise),

- `proxy_set_header Host $host;` directive ensures that the correct `Host` is passed to py4web – here `myapp.example.com`

- **proxy_set_header X-PY4WEB-APPNAME /myapp; directive ensures that py4web (and ombott) knows which app to serve**

    and **also** that this application is domain-mapped – pay specific attention to the slash (`/`) in front of the `myapp` name – it is **required** to ensure correct parsing of URLs on ombott level,

- finally **proxy_pass http://127.0.0.1:8000/myapp$request_uri;** ensures that the request is passed in its integrity (**$request_uri**)

    to py4web server (here: `127.0.0.1:8000`) and the correct app (`/myapp`).

Such configuration ensures that all URL manipulation inside ombott and py4web - especially in modules such as `Auth`, `Form`, and `Grid` are done correctly using the domain to which the app is mapped to.

5.8 Custom error pages

py4web provides default error pages. For instance, if none of the routes in an app matches the request, a default 404 error page will be shown. By default all HTTP error codes are handled automatically by py4web.

It is however possible to override this behaviour. It can be done either per HTTP error code, or even for all errors.

Here is an example for overriding HTTP code 404 (not found):

```
from py4web.core import ERROR_PAGES
ERROR_PAGES[404] = f"Page not found!"
```

If one wants to replace _all_ default error pages, a special qualifier `"*"` should be used. Also, the returned value may contain HTML code as well:

```
from py4web import URL
from py4web.core import ERROR_PAGES
from yatl.helpers import A

ERROR_PAGES["*"] = f"We have encountered an error! (try: {A('Main Page',
_href=URL("/",scheme=True))})"
```

Note that this setup is **global**. This means that it is defined once for all apps on a given py4web instance. This is because, when an error is encountered, it could be because the request has not matched any of the apps. Hence, this configuration should only be done in **one of the apps**.

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 HTTP 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. Think of fixtures as per action (as opposed to per app) middleware.

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: sessions, url signing and flash messages will be fully explained in this chapter. Database connections, internationalization, authentication, and templates will instead be just outlined here since they have dedicated chapters.

The developer is also free to add fixtures, for example, to handle a third party template language or third party session logic; this is explained later in the Section 6.12 paragraph.

6.1 Using Fixtures

As we've seen in the previous chapter, fixtures are the arguments of the decorator `@action.uses(...)`. You can specify multiple fixtures in one decorator or you can have multiple decorators.

Also, fixtures can be applied in groups. For example:

```
preferred = action.uses(session, auth, T, flash)
```

Then you can apply all of them at once with:

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

Usually, it's not important the order you use to specify the fixtures, because py4web knows well how

to manage them if they have explicit dependencies. For example auth depends explicitly on db and session and flash, so you do not even needs to list them.

But there is an important exception: the Template fixture must always be the **first one**. Otherwise, it will not have access to various things it should need from the other fixtures, especially Inject() and Flash() that we'll see later.

6.2 The Template fixture

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

```python
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:

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

Also notice that py4web template files are cached in RAM. The py4web caching object is described later on Section 6.14.

> **Warning** If you use multiple fixtures, always place the template as the **first one**.
>
> For example:
>
> ```python
> @action.uses(session, db, 'index.html') # wrong
> @action.uses('index.html', session, db) # right
> ```
>
> Be careful if you read old documentations that this need was **exactly the opposite** in early py4web experimental versions (until February 2022)!

As we've already seen in the last paragraph, you can combine many fixtures in one decorator. But you can even extend this decorator by passing different templates as needed. For example:

```python
def preferred(template, *optional):
    return action.uses(template, session, auth, T, flash, *optional)
```

And then:

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

This syntax has no performance implications: it's just for avoiding to replicate a decorator logic in multiple places. In this way you'll have cleaner code and if needed you'll be able to change it later in one place only.

6.3 The Inject fixture

The Inject fixture is used for passing variables (and even python functions) to templates. Here is a simple example:

```python
from py4web.utils.factories import Inject
my_var = "Example variable to be passed to a Template"

...

@action.uses('index.html', Inject(my_var=my_var))
def index():

    ...
```

It will be explained later on Section 10.5 in the YATL chapter.

6.4 The Translator fixture

Here is an example of usage:

```python
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".

```python
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))
```

If the `T` fixture is to be used from inside a template you may want to pass it to the template:

```python
@action('index')
```

```
@action.uses("index.html", session, T)
def index():
    return dict(T=T)
```

Or perhaps inject (same effect as above)

```
from py4web.utils.factories import Inject

@action('index')
@action.uses("index.html", session, Inject(T=T))
def index():
    return dict()
```

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"
  }
}
```

Set your browser preference to Italian: now the messages will be automatically translated to Italian.

Notice there is an UI in the Dashboard for creating, updating, and updating translation files. It can be easily reached via the button `i18n+p11n`:
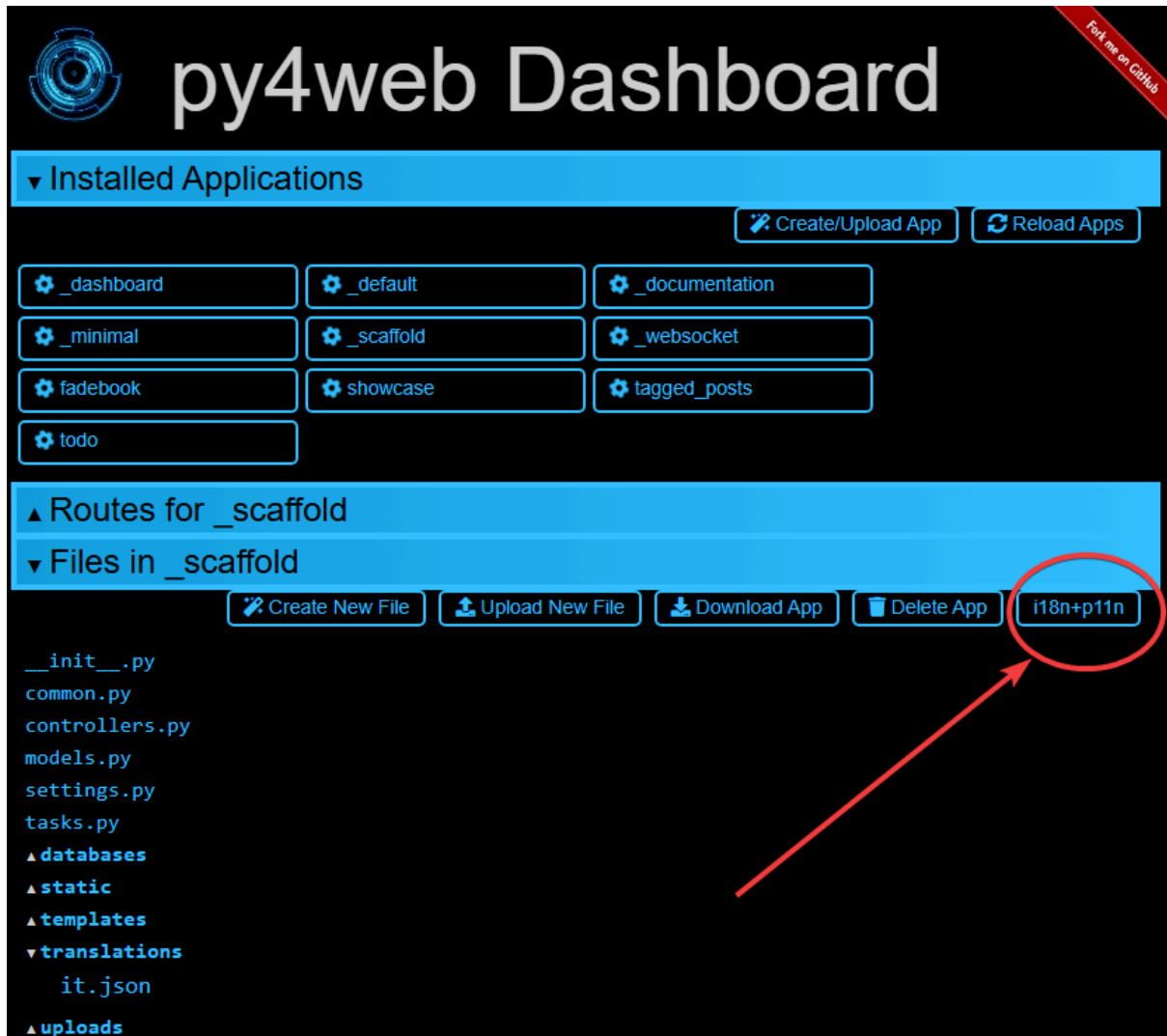
that leads to the following interface:
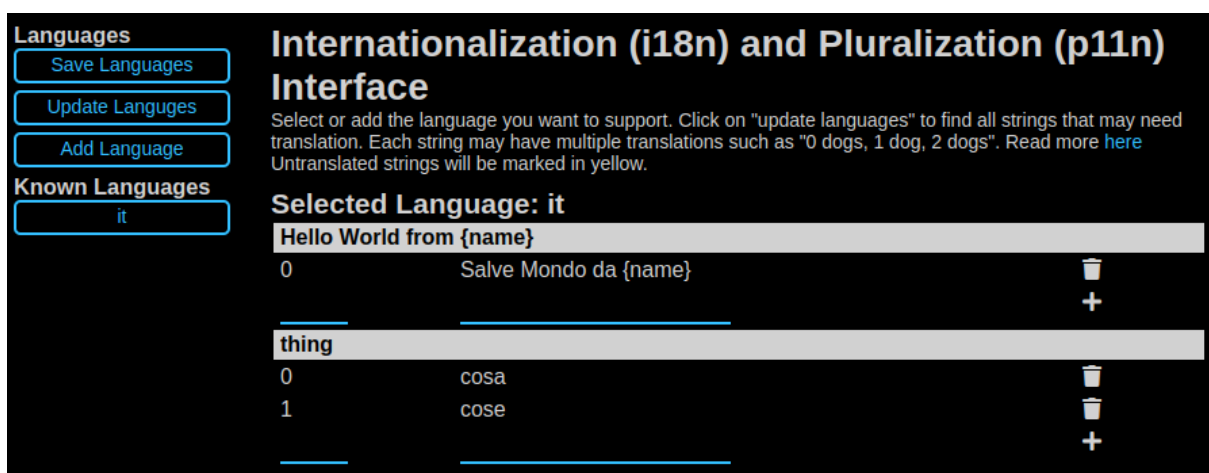
More details can be found here: https://github.com/web2py/pluralize

If you want to force an action to use language defined somewhere else, for example from a session variable, you can do:

```
@action('index')
```

```
@action.uses("index.html", session, T)
def index():
    T.select(session.get("lang", "it"))
    return dict(T=T)
```

If you want all of your action to use the same pre-defined language and ignore browser preferences, you have to redefine the select method for the T instance:

```
T.on_request = lambda *_: T.local.__dict__.update(tag="it",
language=T.languages["it"])
```

This is to be done outside any action and will apply to all actions. Action will still need to declare `action.uses(T)` else the behavior is undefined.

6.5 The Flash fixture

It is common to want to display "alerts" to the users. Here we refer to them as **flash messages**. 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
- can be generated both server side and client side
- may have a type
- 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:

```
<flash-alerts class="padded"
data-alert="[[=globals().get('flash','')]]"></flash-alerts>
```

By setting the value of the message in the flash helper, a flash variable is returned by the action and this triggers 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 the content, unless it is overwritten by another set.

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

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

py4web defaults to an alert class called `info` 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.

You can see the basic usage of flash messages in the **examples** app.

6.6 The Session fixture

Simply speaking, a session can be defined as a way to preserve information that is desired to persist throughout the user's interaction with the web site or web application. In other words, sessions render the stateless HTTP connection a stateful one.

In py4web, the session object is also a fixture. Here is a simple example of its usage to implement a counter.

```python
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
```

The counter will start from 0; its value will be remembered and increased every time you reload the page.
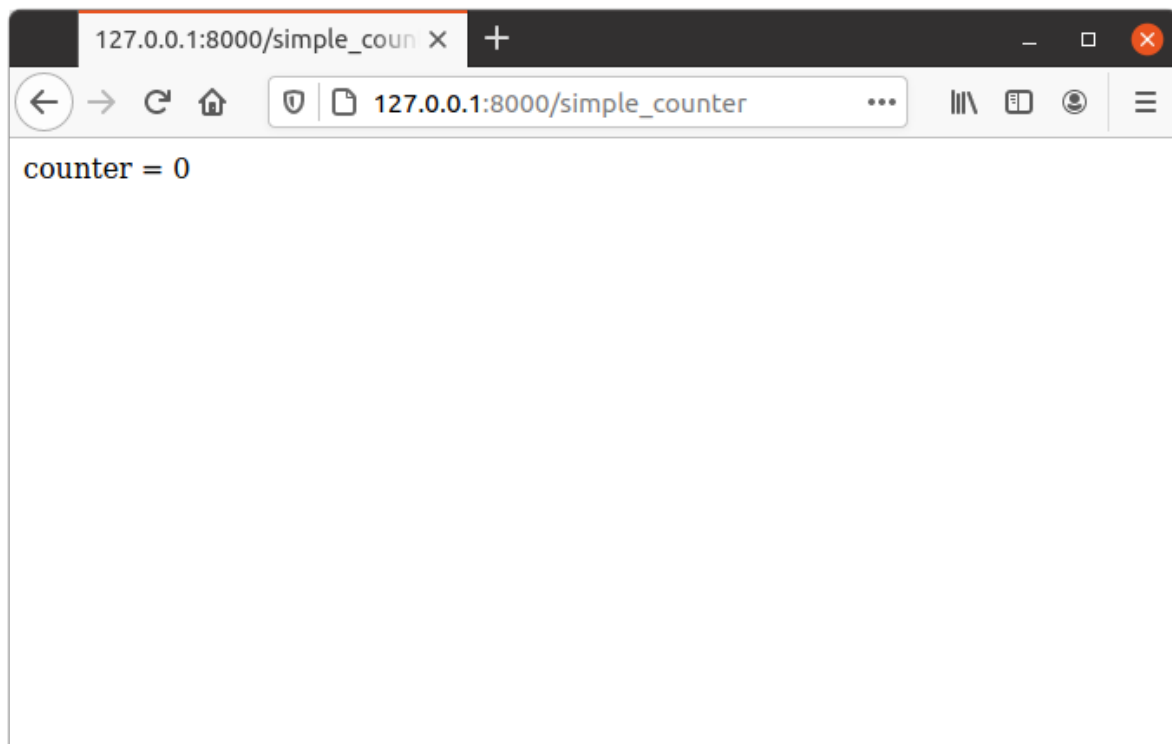
Opening the page in a new browser tab will give you the updated counter value. Closing and reopening the browser, or opening a new *private window*, will instead restart the counter from 0.

Usually the information is saved in the session object are related to the user - like its username, preferences, last pages visited, shopping cart and so on. The session object has the same interface as a Python dictionary but in py4web sessions are always stored using JSON (**JWT** specifically, i.e. JSON Web Token), 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.

The information composing the session object can be saved:

- client-side, by only using cookies (default)
- server-side, but you'll still need minimal cookies for identifying the clients

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:

```python
session = Session(secret='my secret key',
                  expiration=3600,
                  algorithm='HS256',
                  storage=None,
                  same_site='Lax',
                  name="{app_name}_sesson")
```

Here:

- `secret` is the passphrase used to sign the information
- `expiration` is the maximum lifetime of the session, in seconds (default = None, i.e. no timeout)
- `algorithm` is the algorithm to be used for the JWT token signature ('HS256' by default)
- `storage` is a parameter that allows to specify an alternate session storage method (for example Redis, or database). If not specified, the default cookie method will be used
- `same_site` is an option that prevents CSRF attacks (Cross-Site Request Forgery) and is enabled by default with the 'Lax' option. You can read more about it here
- `name` is the format to use for the session cookie name.

If storage is not provided, session is stored in client-side jwt cookie. Otherwise, we have server-side session: the jwt is stored in storage and only its UUID key is stored in the cookie. This is the reason why the secret is not required with server-side sessions.

6.6.1 Client-side session in cookies

By default the session object is stored inside a cookie called `appname_session`. It's a JWT, hence encoded in a URL-friendly string format and signed using the provided secret for preventing tampering.

> **Warning** Data embedded in cookies is signed, not encrypted! In fact it's quite trivial to read its content from http communications or from disk, so do not place any sensitive information inside, and use a complex secret.

If the secret changes existing sessions are invalidated. If the user switches from HTTP to HTTPS or vice versa, the user session is also invalidated. Session in cookies have a small size limit (4 kbytes after being serialized and encoded) so do not put too much into them.

6.6.2 Server-side session in memcache

Requires memcache installed and configured.

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

6.6.3 Server-side session in Redis

Requires Redis installed and configured.

```python
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.

6.6.4 Server-side session in database

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

> **Warning** the `'sqlite:memory'` database used in this example **cannot be used in multiprocess environment**; the quirk is that your application will still work but in non-deterministic and unsafe mode, since each process/worker will have its own independent in-memory database.

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

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

6.6.5 Server-side 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:

```python
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 recommend storing sessions on the filesystem: it is inefficient and does not scale well.

6.6.6 Sharing sessions

Imagine you have an app "app1" which uses a session and an app "app2" that wants to share a session with app1. Assuming they use sessions in cookies, "app2" would use:

```python
session = Session(secret=settings.SESSION_SECRET_KEY,
                  name="app1_session")
```

The name tells app2 to use the cookie "app1_session" from app1. Notice it is important that the secret

---

is the same as app1's secret. If using a session in db, then app2 must be using the same db as app1. It is up to the user to make sure that the data stored in the session and shared between the two apps are consistent and we strongly recommend that only app1 writes to the session, unless the share one and the same database.

Notice that it is possible for one app to handle multiple sessions. For example one session may be its own, and another may be used exclusively to read data from another app (app1) running on the same server:

```
session_app1 = Session(secret=settings.SESSION_SECRET_KEY,
                       name="app1_session")
...
@action.uses(session, session_app1)
...
```

6.7 The Condition fixture

Sometimes you want to restrict access to an action based on a given condition. For example to enforce a workflow:

```
@action("step1")
@action.uses(session)
def step1():
    session["step_completed"] = 1
    button = A("next", _href=URL("step2"))
    return locals()

@action("step2")
@action.uses(session, Condition(lambda: session.get("step_completed") == 1))
def step2():
    session["step_completed"] = 2
    button = A("next", _href=URL("step3"))
    return locals()

@action("step3")
@action.uses(session, Condition(lambda: session.get("step_completed") == 2))
def step3():
    session["step_completed"] = 3
    button = A("next", _href=URL("index"))
    return locals()
```

Notice that the Condition fixtures takes a function as first argument which is called `on_request` and must evaluate to True or False.

Also notice that in the above example the Condition depends on the Session therefore it must be listed after `session` in `action.uses`.

If False, by default, the Condition fixture raises 404. It is possible to specify a different exception:

```
Condition(cond, exception=HTTP(400))
```

It is also possible to call a function before the exception is raised, for example, to redirect to another page:

```
Condition(cond, on_false=lambda: redirect(URL('step1')))
```

You can use condition to check permissions. For example, if you are giving group memberships to users using `Tags` (it will be explained later on the Section 13.2 chapter), then you can require that users action have specific group membership:

```
groups = Tags(db.auth_user)
```

```
@action("payroll")
@action.uses(auth,
             Condition(lambda: 'employees' in groups.get(auth.user_id),
on_false=lambda: redirect('index')))
def payroll():
    return
```

6.8 The URLsigner fixture

A signed URL is a URL that provides limited permission and time to make an HTTP request by containing authentication information in its query string. The typical usage is as follows:

```
from py4web.utils import URLSigner

# We build a URL signer.
url_signer = URLSigner(session)

@action('/somepath')
@action.uses(url_signer)
def somepath():
    # This controller signs a URL.
    return dict(signed_url = URL('/anotherpath', signer=url_signer))

@action('/anotherpath')
@action.uses(url_signer.verify())
def anotherpath():
    # The signature has been verified.
    return dict()
```

6.9 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 the next 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.

6.10 The Auth fixture

`auth` and `auth.user` are both fixtures that depend on `session` and `db`. Their role is to provide the action with authentication information.

Auth is used as follows:

```python
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).

If a `auth_user` table is defined before calling `auth.enable()` the provided table will be used.

It is also possible to add `extra_fields` to the `auth_user` table, for example:

```python
extra_fields = [
    Field("favorite_color"),
]
auth = Auth(session, db, extra_fields=extra_fields)
```

In any case, we recommend not to pollute the `auth_user` table with extra fields but, instead, to use one of more additional custom tables that reference users and store the required information.

The `auth` object exposes the method:`auth.enable()` which registers multiple actions including `{appname}/auth/login`. It requires the presence of the `auth.html` template and the `auth` value component provided by the `_scaffold` app. It also exposes the 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` and in this case the code of the example redirects to the `auth/login` page.

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

```python
@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, hence this example is equivalent to the previous one.

The `auth` fixture is plugin based: it supports multiple plugin methods including OAuth2 (Google, Facebook, Twitter), PAM and LDAP. The Chapter 13 chapter will show you all the related details.

6.11 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:

```python
from py4web import 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('generic.html', db)
def index():
    db.thing.name.writable = True
    form = Form(db.thing)
    return dict(form=form)
```

Note that 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.

6.12 Custom fixtures

A fixture is an object with the following minimal structure:

```python
from py4web.core import Fixture


class MyFixture(Fixture):
    def on_request(self, context): pass
    def on_success(self, context): pass
    def on_error(self, context) pass
```

For example in the DAL fixture case, `on_request` starts a transaction, `on_success` commits it, and `on_error` rolls it back.

In the case of a template, `on_request` and `on_error` do nothing but `on_success` transforms the output.

In the case of `auth.user` fixtures, `on_request` does all the work of determining if the user is logged in (from the dependent session fixture) and eventually preventing the request from accessing the inner layers.

Now imagine a request coming in calling an action with three fixtures A, B, and C. Under normal circumstances above methods are executed in this order:

```
request  -> A.on_request -> B.on_request -> C.on_request -> action
response <- A.on_success <- B.on_success <- C.on_success <-
```

i.e. the first fixture (A) is the first one to call `on_request` and the last one to call `on_success`. You

can think of them as layers of an onion with the action (user code) at the center. `on_success` is called when entering a layer from the outside and `on_success` is called when exiting a layer from the inside (like WSGI middleware).

If any point an exception is raised inner layers are not called and outer layers will call `on_error` instead of `on_success`.

Context is a shared object which contains:

- content['fixtures']: the list of all the fixtures for the action.

- context['processed']: the list of fixtures that called `on_request` previously within the request.

- context['exception']: the exception raised by the action or any previous fixture logic (usually None)

- context['output']: the action output.

`on_success` and `on_error` can see the current `context['exception']` and transform it. They can see the current `context['output']` and transform it as well.

For example here is a fixture that transforms the output text to upper case:

```python
class UpperCase(Fixture):
    def on_success(self, context):
        context['output'] = context['output'].upper()

upper_case = UpperCase()

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

Notice that this fixture assumes the `context['output']` is a string and therefore it must come before the template.

Here is a fixture that logs exceptions tracebacks to a file:

```python
class LogErrors(Fixture):
    def __init__(self, filename):
        self.filename = filename
    def on_error(self, context):
        with open(self.filename, "a") as stream:
            stream.write(str(context['exception']) + '\n')

errlog = LogErrors("myerrors.log")

@action('index')
@action.uses(errlog)
def index(): return 1/0
```

Fixtures also have a __prerequisite__ attribute. If a fixture takes another fixture as an argument, its value must be appended to the list of __prerequisites__. This guarantees that they are always executed in the proper order even if listed in the wrong order. It also makes it optional to declare prerequisite fixtures in `action.uses`.

For example `Auth` depends on `db`, `session`, and `flash`. `db` and `session` are indeed arguments. `flash` is a special singleton fixture declared within `Auth`. This means that

```
action.uses(auth)
```

is equivalent to

```
action.uses(auth, session, db, flash)
```

Why are fixtures not simply functions that contain a try/except?

We considered the option but there are some special exceptions that should not be considered errors but success (`py4web.HTTP`, `bottle.HTTResponse`) while other exceptions are errors. The actual logic can be complicated and individual fixtures do not need to know these details.

They all need to know what the context is and whether they are processing a new request or a response and whether the response is a success or an error. We believe this logic keeps the fixtures easy.

Fixtures should not in general communicate with each other but nothing prevents one fixture to put data in the context and another fixture to retrieve that data.

6.12.1 Fixtures with dependencies

If a fixture depends on another fixture, it needs to be passed that fixture in the initializer, and the fixture must be listed in the `__prerequisites__` attribute. For example, suppose we want to create a fixture that grants access to a controller only to users whose email address is included in an ADMIN_EMAILS list. We can write the following fixture:

```python
class AdminAccess(Fixture):

    def __init__(self, auth, admin_list, redirect_url=None):
        super().__init__()
        self.admin_list = admin_list
        self.auth = auth
        self.__prerequisites__ = [auth]
        # One thing to note here is that the URL function can only be called in a
        # request context (while serving a request).  Thus, we cannot store in
the fixture
        # initialization the full URL to redirect, but only the path.
        self.redirect_url = redirect_url or 'index'

    def on_request(self, context):
        if ((not self.auth.current_user)
                or self.auth.current_user.get('email') not in self.admin_list):
            redirect(URL(self.redirect_url))

    def on_error(self, context):
        redirect(URL(self.redirect_url))
```

The fixture can be created and used as follows:

```python
admin_access = AdminAccess(auth, ['a@example.com',], 'index')

@action('/admin-only')
@action.uses('admin_only.html', admin_access)
def admin_only():
    return dict()
```

6.12.2 Using local storage

Fixtures can use a thread-local storage for data they need. Here is an example:

```python
class LocalStorageDemo(Fixture):

    def __init__(self):
        super().__init__()

    def on_request(self, context):
        Fixture.local_initialize(self)
        # We can check whether the local storage is valid.
        print(f"is_valid: {self.is_valid()}")
        content = str(uuid.uuid4())
        print(f"Storing content: {content}")
```

```
        self.local.my_content = content

    def on_success(self, context):
        # The line below is used only to show that the thread-local object is in
place.
        print(f"Retrieved: {self.local.my_content}")
```

Notably, the initializer should contain the line:

```
Fixture.local_initialize(self)
```

in order to initialize the thread-local storage. Once this is done, the thread-local storage can be used to store and retrieve data using the the `self.local` object.

6.13 Multiple fixtures

As previously stated, it's generally not important the order you use to specify the fixtures but it's mandatory that you always place the template as the **first one**. Consider this:

```
@action("index")
@action.uses(A, B)
def func(): return "Hello world"
```

Pre-processing (`on_request`) in the fixtures happen in the sequence they are listed and then the `on_success` or `on_error` methods will be executed in reverse order (as an onion).

Hence the previous code can be explicitly transformed to:

```
A.on_request()
B.on_request()
func()
B.on_success()
A.on_success()
```

So if A.on_success() is a template and B is an inject fixture that allows you to add some extra variables to your templates, then A must come first.

Notice that

```
@action.uses(A)
@action.uses(B)
```

is almost equivalent to

```
@action.uses(A, B)
```

but not quite. All fixtures declared in one `action.uses` share the same context while fixtures in different `action.uses` use different contexts and therefore they cannot communicate with each other. This may change in the future. For now we recommend using a single call to `action.uses`.

6.14 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` decorator 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.

6.15 Convenience Decorators

The `_scaffold` application, in `common.py` defines two special convenience decorators using Action-Factory:

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

and

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

They apply all of the decorators below (db, session, T, flash, auth), use a template with the same name as the function (.html), and also register a route with the name of action followed by 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.

Warning ActionFactory decorators like these cannot be combined with @action or @action.uses

---

The decorators can be used directly as shown above, which enables all HTTP methods (GET, POST, PUT, …) but you can also create separate controllers for each HTTP method:

```
@authenticated.get()
def index():
    # only handle GET requests
    return dict()


@authenticated.post(path="index")
def index_form():
    # only handle POST requests
    return dict()
```

The both decorator and its HTTP method calls have the following arguments:

-
  path overwrites the path built from the function name
      with the given string. Does not automatically handle arguments.

-
  template specifies the template name, instead of using
      the function name.

- `uses` specify extra fixtures for this specific controllers.

```
@authenticated(
    path="test",
    template="generic.html",
    uses=[Inject(message="Hello World")])
def example():
    return dict()
```

As manual ordering of fixtures isn't possible with `uses`, make sure the fixtures define their dependencies. See: Section 6.12.1

The Database Abstraction Layer (DAL)

7.1 DAL introduction

py4web rely on 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. The DAL choosen is a pure Python one called pyDAL. It was conceived in the web2py project but it's a standard python module: you can use it in any Python context.

Note What makes pyDAL different from most of the other DALs is the syntax: it maps records to python dictionaries, which is simpler and closer to SQL. Other famous frameworks instead strictly rely on an Object Relational Mapping (**ORM**) like the Django ORM or the SQL Alchemy ORM, that maps tables to Python classes and rows to Objects.

A little taste of pyDAL features:

- Transactions
- Aggregates
- Inner & Outer Joins
- Nested Selects

Note An important difference between py4web and web2py is that only some few Field attributes are safe to modify in actions. See Section 7.5.1 for more info, and Chapter 15 for a general list of differences.

7.1.1 Supported databases

A partial list of supported databases is show in the table below. Please check on the py4web/pyDAL web site and mailing list for more recent adapters.

Note In any modern python distribution **SQLite** is actually built-in as a Python library. The SQLite driver (sqlite3) is also included: you don't need to install it. Hence this is the most popular database for testing and development.

The Windows and the Mac binary distribution work out of the box with SQLite and PostgreSQL only. To use any other database back end, run a full py4web distribution and install the appropriate driver for the required back end. Once the proper driver is installed, start py4web and it will automatically

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

Support of MongoDB is experimental. Google NoSQL is treated as a particular case. The Section 7.15 section at the end of this chapter has some more information about specific databases.

7.1.2 The DAL: a quick tour

py4web defines the following classes that make up the DAL:

DAL

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` does.

```
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`.

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)
```

7.1.3 Using the DAL "stand-alone"

pyDAL is an independent python package. As such, it can be used without the web2py/py4web environment; you just need to install it with `pip`. Then import the pydal module when needed:

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

Note Even if you can import modules directly from pydal, this is not advisable from within py4web applications. Remember that `py4web.DAL` is a fixture, `pydal.DAL` is not. In this context, the last command should better be:

```
>>> from py4web import DAL, Field
```

7.1.4 Experiment with the py4web shell

You can also experiment with the pyDAL API using the py4web shell, that is available using the Section 3.6.6.

> **Warning** Mind that database changes may be persistent. So be careful and do NOT hesitate to create a new application for doing testing instead of tampering with an existing one. The only exception is the showcase db: in case of problems you can recreate it by simply deleting the database folder and restarting py4web. This will re-create the database with all the example data.

Note that most of the code snippets that contain the python prompt `>>>` are also directly executable via a py4web shell.

This is a simple example, using the provided `showcase` app:

```
>>> from apps.showcase.examples.models import db
>>> db.tables()
['auth_user', 'auth_user_tag_groups', 'person', 'superhero', 'superpower', 'tag',
'thing', 'user_token', 'dummy']
>>> rows = db(db.superhero.name != None).select()
>>> rows.first()
<Row {'id': 1, 'tag': <Set ("tag"."superhero" = 1)>, 'name': 'Superman',
'real_identity': 1}>
```

You can also start by creating a connection from zero. For the sake of simplicity, you can use SQLite. Nothing in this discussion changes when you switch the back-end engine.

7.2 Using the dashboard app with databases

Generally you can use the dashboard app for viewing and modifying the databases of a particular

---

app. However this is not bulletproof, so for security reason this by default is not applied to the show-case app. But if your installation is local (not exposed to public networks), you can enable it by simply adding to the file``apps/showcase/__init__.py`` the line:

```
from .examples.models import db
```

This allow you to look graphically inside the showcase application database:
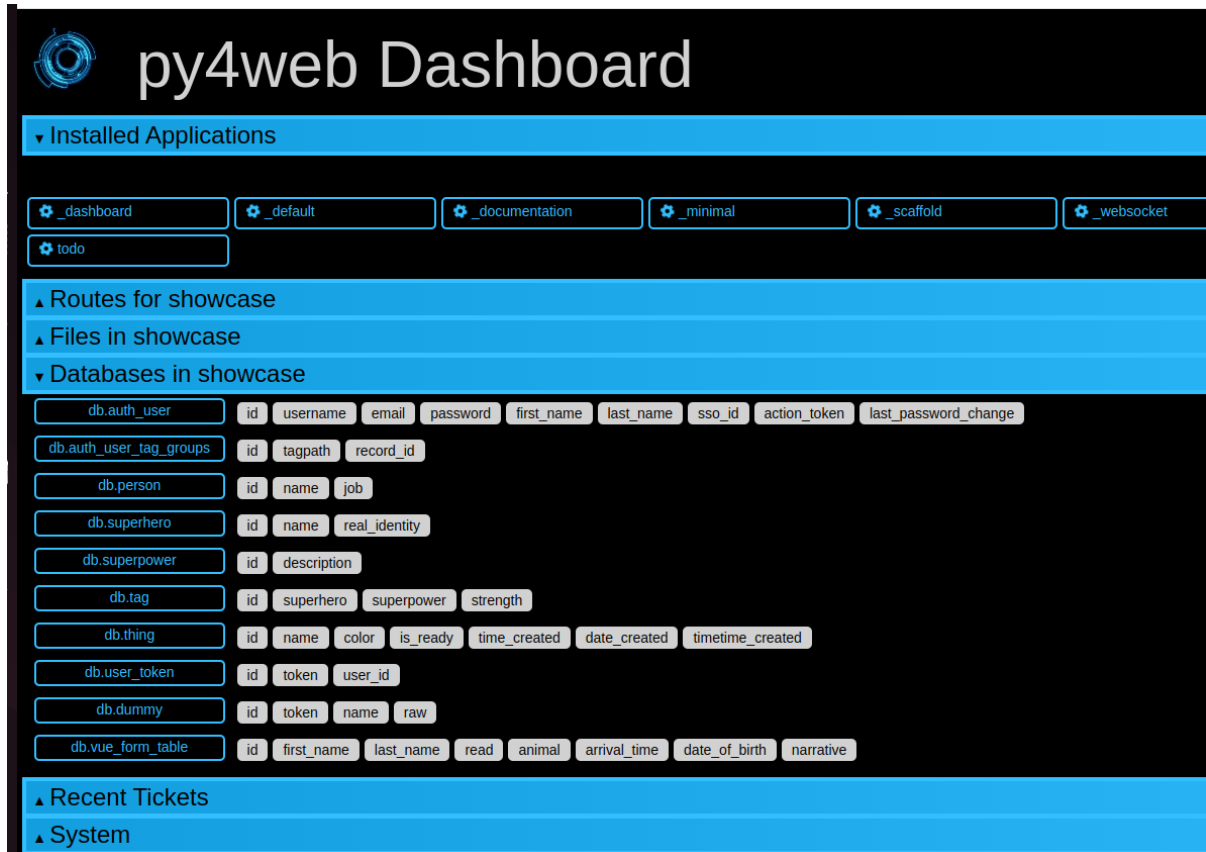
7.3 DAL constructor

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 `_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 data-bases of different types. For now, we will assume the presence of a single database since this is the most common situation.

7.3.1 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)
```

7.3.2 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')
```

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"):

Database	Connection string
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`

- 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 addition to the file 'storage.sqlite' that contains the data, there will be also a sql.log file plus one additional file called longhash_tablename.table for every table definition. The table definition files are used during migrations; in case of problems they could be deleted (they'll be automatically recreated).

- 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 also 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; see Section 7.8.5 for details. 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.

7.3.3 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 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.

7.3.4 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.

7.3.5 Lazy Tables

Setting `lazy_tables = True` provides a major performance boost (but not with py4web). It means that table creation is deferred until the table is actually referenced.

> **Warning** You should never use lazy tables in py4web. There is no advantage, no need, and possibly concurrency problems.

7.3.6 Model-less applications

Normally in py4web the code that define DAL tables lives in the file `models.py`, hence it's only executed at startup because it's outside of actions.

However, it is possible to define DAL tables on demand inside actions. This is referred to as "model-less" development by the py4web community.

To use the "model-less" approach, you take responsibility for doing all the housekeeping tasks. You call the table definitions when you need them, and provide database connection passed as parameter. Also, remember maintainability: other py4web developers expect to find database definitions in the `models.py` file.

7.3.7 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.

7.3.8 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 db connection 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.

Database	check_reserved
PostgreSQL	`postgres(_nonreserved)`
MySQL	`mysql`
FireBird	`firebird(_nonreserved)`
MSSQL	`mssql`
Oracle	`oracle`

7.3.9 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
```

7.3.10 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.

7.3.11 Other DAL constructor parameters

Database folder location

`folder` sets the place where migration files will be created (see Section 7.6 for details). By default it's automatically set within py4web on the same folder of the database itself, but you have to specify it when using DAL outside py4web.

Note that for SQLite databases it's normally necessary, otherwise you'll implicitly choose an in memory database (where folder and migrations don't have any sense). So these constructors have the same meaning:

```
db = DAL('sqlite://storage.sqlite') # folder parameter not specified
```

```
db = DAL('sqlite:memory')              # in memory database
```

Default migration settings

The DAL constructor migration settings are booleans affecting defaults and global behaviour (again, see Section 7.6 for details)

`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

7.3.12 `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.

If you pass `db` in an `action.uses` decorator, you don't need to call commit in the controller, it is automatically done for you (also, if you use `authenticated` or `unauthenticated` decorator.)

Tip always add `db` in an `action.uses` decorator (or use the `authenticated` or `unauthenticated` decorator). Otherwise you have to add `db.commit()` in every define_table and in every table activities: insert(), update(), delete()

So in actions there is normally no need to ever call `commit` or `rollback` explicitly in py4web unless you need more granular control.

But 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
```

7.4 Table constructor

Tables are defined in the DAL via `define_table`.

7.4.1 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.

7.4.2 `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.

7.4.3 `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.

7.4.4 `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.

7.4.5 `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.

7.4.6 `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'`

7.4.7 `primarykey`: Support for legacy tables

`primarykey` helps support legacy tables with existing primary keys, even multi-part. See Section 7.4.15.

7.4.8 `migrate, fake_migrate`

`migrate` sets migration options for the table. Refer to Section 7.6 for details.

7.4.9 `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)
```

7.4.10 `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.

7.4.11 `trigger_name`

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

7.4.12 `polymodel`

For use with Google App Engine.

7.4.13 `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`.

7.4.14 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 attribute 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 = {}
```

7.4.15 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.

7.5 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 instead it is used by `Form` (this will be explained better on the Chapter 12 chapter). The default validators for the given types are shown in the next section Section 7.5.2.

  Note 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 Section 7.5.4.

- `uploadfolder` must be set to a location where to store uploaded files. The scaffolding app defines a folder `settings.UPLOAD_FOLDER` which points to `apps/{app_name}/uploads` so you can set, for example, `Field(... uploadfolder=settings.UPLOAD_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. pydal either uses the separate subfolders or it does not. Changing the behavior after files have been uploaded will prevent pydal 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.

- `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. 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 autogenerated 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.

- `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.

- `widget` Do NOT use the widget parameter in py4web for a Field definition. (This was a feature of web2py and is not to be used in py4web) See Section 12.4 later.

- `represent` can be None or can point to a function that takes a field value and returns an alternate representation for the field value.

7.5.1 Thread safety and Field attributes

Even though py4web and web2py use the same pyDAL, there is an important difference which stems from the core architecture of py4web. In py4web only the following `Field` attributes can be changed inside an action:

- `readable`

- `writable`
- `default`
- `filter_in`
- `filter_out`
- `label`
- `update`
- `requires`
- `widget`
- `represent`

These are reset to their original values before each action is called. All other `Field`, `DAL`, and `Table` attributes are global and non-thread-safe.

This limitation exists because py4web executes table definitions only at startup, unlike web2py which re-defines tables on each request. This makes py4web a lot faster than web2py, but you need to be careful as modifying non-thread-safe attributes in actions can cause race conditions and bugs.

7.5.2 Field types and validators

Type	Default 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(db, 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 Section 7.14.1.

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 advantage of making the communication independent of the back-end specific escaping conventions.

7.5.3 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.

7.5.4 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)
```

7.6 Migrations

With our example table definition:

```
db.define_table('person')
```

`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".

Note by default py4web uses the "app/databases" folder for the log file and all other migration files it needs. You can change this setting by changing 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'))`

The first argument of `define_table` is always the table name. The other unnamed arguments are the fields. 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.

Note 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 disable them.

7.6.1 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 "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.

7.6.2 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
```

7.7 Table methods

7.7.1 `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.

7.7.2 `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.

7.7.3 `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')
```

7.7.4 `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"]`.

7.7.5 `drop`

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

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

7.7.6 QueryBuilder

You can generate DAL queries using natural language. This can be done as follows:

```python
from py4web import DAL, Field
from pydal import QueryBuilder, QueryParseError

db = DAL("sqlite:memory")
db.define_table("thing", Field("name"), Field("solid", "boolean"))
builder = QueryBuilder(db.thing)
query = builder.parse('name is equal to Chair')
rows = db(query).select()
```

Example of valid expressions are:

- name is null
- name is not null
- name == Chair
- name is Chair
- name is equal Chair
- name is equal to Chair
- name is equal to "Chair"
- name lower is equal to Chair
- not name lower is equal to Chair
- not (name lower is equal to Chair)
- name == Chair or name == Table
- name starts with C and name contains air
- name in Chair, Table, Glass
- name belongs Chair, Table, "Glass Top"

- solid is true

- solid is false

Notice that quotes are optional and only for values. You can use brakets to group. It throws a Query-ParseError exception on failure. You can pass optional parameters to QueryBuilder for internationalization purposes (Italian in example):

```
builder = QueryBuilder(db.thing,
            field_aliases={"id": "id", "nome": "name"},
            token_aliases={"non è nullo": "is not null", "è uguale a": "=="})
query = builder.parse('nome non è nullo')
query = builder.parse('nome è uguale a Tavolo')
```

If field_aliases is not provided, only readable fields can be searched. If field_aliases is provided, only explicitedly included field names can be searched.

You can, of course, use the translation operator T:

```
builder = QueryBuilder(db.thing,
            field_aliases={str(T(field.name)): field name for field in db.thing},
            token_aliases={str(T(key):key) for key in QueryBuilder.token_ops})
```

You can alias the following tokens:

```
{
    # boolean tokens
    "not",
    "and",
    "or",
    # field transformers
    "upper",
    "lower",
    # unary search expressions
    "is null",
    "is not null",
    "is true",
    "is false",
    # binary search expressions
    "==",
    "!=",
    "<",
    ">",
    "<=",
    ">=",
    "contains",
    "startswith", # notice "starts with" is an alias!
    # search in list
    "belongs",
}
```

The query builder is used in the Grid. In the Grid the field aliases are the field.label in lower case with the spaces replaced by underscore.

7.7.7 Tagging records

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

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

db = DAL("sqlite:memory")
db.define_table("thing", Field("name"))
id1 = db.thing.insert(name="chair")
```

```
id2 = db.thing.insert(name="table")
properties = Tags(db.thing)
properties.add(id1, "color/red")
properties.add(id1, "style/modern")
properties.add(id2, "color/green")
properties.add(id2, "material/wood")

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

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

rows = db(properties.find(["material/wood"])).select()
assert rows.first().id == id2

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

`Tags` are hierarchical. Then `find(["color"])` would return id1 and id2 because both records have tags with "color".

It is internally implemented with the creation of an additional table, which in this example would be db.thing_tags_default, because no tail was specified on the `Tags(table, tail="default")` constructor.

py4web uses `Tags` as a flexible mechanism to manage permissions, we'll see all the details later on the Section 13.2 chapter.

7.8 Raw SQL

7.8.1 `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.

7.8.2 `_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.

7.8.3 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.

7.8.4 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.

7.8.5 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.

7.9 `select` command

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. 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 an 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")
```

7.9.1 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.

7.9.2 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.

7.9.3 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.

7.9.4 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`.

7.9.5 Recursive `select`s

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.

7.9.6 `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).

```
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 Section 7.11.1. They are described in Section 7.11.2 and Section 7.11.3 sections respectively.

cache, cacheable

An example use which gives much faster selects is:

```
rows = db(query).select(cache=(cache.get, 3600), cacheable=True)
```

Look at Section 7.9.17, to understand what the trade-offs are.

7.9.7 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'
```

7.9.8 `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.

7.9.9 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()
```

7.9.10 `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
```

7.9.11 `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': 2, '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': 3, '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=datetime.datetime.utcnow)`). 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`.

7.9.12 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".

7.9.13 `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.

7.9.14 `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 (Rows objects themselves cannot be serialized because they contain a reference to an open DB connec-

tion):connection):

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

7.9.15 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
```

7.9.16 `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.

7.9.17 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 function with signature *(key, callback, expiration)* (for example `cache.get` assuming `cache` is an instance of the py4web cache object), 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.get, 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.get, 3600), cacheable=True)
```

7.10 Computed and Virtual fields

7.10.1 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': 5}>
```

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.

7.10.2 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.

7.10.3 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`.

7.10.4 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:
    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:
    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:
    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:
    def discounted_unit_price(self):
        return self.item.unit_price * 0.90

class MyVirtualFields2:
    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:
    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:
    def lazy_total_price(self):
        return lambda self=self: self.item.unit_price * self.item.quantity
```

7.11 Joins and Relations

7.11.1 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')` or: `Field('owner_id', db.person)`.

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
```

7.11.2 Inner join

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.

7.11.3 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.

7.11.4 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.

7.11.5 Many to many relation

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, see the Section 13.2 chapter. Tagging works
even on database backends that do not support JOINs like the Google App Engine NoSQL.

7.11.6 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.

7.12 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
```

7.12.1 `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 Section 7.14.1.

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
```

7.12.2 `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
```

7.12.3 `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.

7.12.4 `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
```

7.12.5 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])
```

7.12.6 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'))
```

7.13 Exporting and importing data

7.13.1 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()))
```

Converting a `Rows` object into a string produces an encoded binary string and it's better to be explicit with the encoding used:

```
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, 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.

7.13.2 CSV (all tables at once)

In py4web, you can backup/restore an entire database with two commands:

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.

7.13.3 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 datetime
import uuid

now = datetime.datetime.utcnow

db.define_table('person',
                Field('uuid', length=64),
                Field('modified_on', 'datetime', default=now, update=now),
                Field('name'))

db.define_table('thing',
                Field('uuid', length=64),
                Field('modified_on', 'datetime', default=now, update=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:

```
from py4web import response

def export():
    s = StringIO.StringIO()
```

```
    db.export_to_csv_file(s)
    response.set_header('Content-Type', 'text/csv')
    return s.getvalue()
```

Create a controller action to import a saved copy of the other database and sync records:

```python
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.

7.13.4 HTML and XML (one Table at a time)

Rows objects also have an `xml` method (like helpers) that serializes it to XML/HTML:

```python
>>> rows = db(db.person.id == db.thing.owner_id).select()
>>> print(rows.xml())
```

```html
<table>
<thead>
<tr><th>person.id</th><th>person.name</th>
    <th>thing.id</th><th>thing.name</th>
    <th>thing.owner_id</th>
</tr>
</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` XML helper that we'll see later 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>
```

> **Warning** Do not confuse the *TAG* XML helper used here (see the Section 10.3.1 chapter) with the `Tags` method that will be extensively explained on the Section 13.2 chapter.

7.13.5 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

7.14 Advanced features

7.14.1 `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

---

7.14. Advanced features   **105**

want to use them in forms.

7.14.2 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:

```
now = datetime.datetime.utcnow

signature = db.Table(db, 'signature',
                     Field('is_active', 'boolean', default=True),
                     Field('created_on', 'datetime', default=now),
                     Field('created_by', db.auth_user, default=auth.user_id),
                     Field('modified_on', 'datetime', update=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.

7.14.3 `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 Section 7.14.7.

7.14.4 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 by 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'}>, 1)
1
>>> 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 callback will ever be called as consequence of a cascade-deletion.

7.14.5 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.

7.14.6 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.

7.14.7 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 pydal 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 because can make your code not portable across database engines.

It does not work on Google App Engine NoSQL.

7.14. Advanced features

7.14.8 Using DAL without define tables

The DAL can be used from any Python program simply by doing this:

```python
from pydal import DAL, Field
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:

```python
from py4web import DAL, Field
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.

7.14.9 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:

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

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

```python
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.

7.14.10 Copy data from one db into another

Consider the situation in which you have been using the following database:

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

and you wish to move to another database using a different connection string:

```python
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.

7.15 Gotchas

7.15.1 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 module "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.

pyDAL 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:

```python
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:

```python
db = DAL('mysql://...')
```

the prefix in the uri string defines the adapter. The mapping is defined in the following dictionary also in "dal.py":

couchdb	pydal.adapters.couchdb.CouchDB
cubrid	pydal.adapters.mysql.Cubrid
db2:ibm_db_dbi	pydal.adapters.db2.DB2IBM
db2:pyodbc	pydal.adapters.db2.DB2Pyodbc
firebird	pydal.adapters.firebird.FireBird
firebird_embedded	pydal.adapters.firebird.FireBirdEmbedded
google:MySQLdb	pydal.adapters.google.GoogleMySQL
google:datastore	pydal.adapters.google.GoogleDatastore
google:datastore+ndb	pydal.adapters.google.GoogleDatastore
google:psycopg2	pydal.adapters.google.GooglePostgres

google:sql	pydal.adapters.google.GoogleSQL
informix	pydal.adapters.informix.Informix
informix-se	pydal.adapters.informix.InformixSE
ingres	pydal.adapters.ingres.Ingres
ingresu	pydal.adapters.ingres.IngresUnicode
jdbc:postgres	pydal.adapters.postgres.JDBCPostgre
jdbc:sqlite	pydal.adapters.sqlite.JDBCSQLite
jdbc:sqlite:memory	pydal.adapters.sqlite.JDBCSQLite
mongodb	pydal.adapters.mongo.Mongo
mssql	pydal.adapters.mssql.MSSQL1
mssql2	pydal.adapters.mssql.MSSQL1N
mssql3	pydal.adapters.mssql.MSSQL3
mssql3n	pydal.adapters.mssql.MSSQL3N
mssql4	pydal.adapters.mssql.MSSQL4
mssql4n	pydal.adapters.mssql.MSSQL4N
mssqln	pydal.adapters.mssql.MSSQL1N
mysql	pydal.adapters.mysql.MySQL
oracle	pydal.adapters.oracle.Oracle
postgres	pydal.adapters.postgres.Postgre
postgres2	pydal.adapters.postgres.PostgreNew
postgres2:psycopg2	pydal.adapters.postgres.PostgrePsycoNew
postgres3	pydal.adapters.postgres.PostgreBoolean
postgres3:psycopg2	pydal.adapters.postgres.PostgrePsycoBoolean
postgres:psycopg2	pydal.adapters.postgres.PostgrePsyco
pytds	pydal.adapters.mssql.PyTDS
sapdb	pydal.adapters.sap.SAPDB
spatialite	pydal.adapters.sqlite.Spatialite
spatialite:memory	pydal.adapters.sqlite.Spatialite
sqlite	pydal.adapters.sqlite.SQLite
sqlite:memory	pydal.adapters.sqlite.SQLite
sybase	pydal.adapters.mssql.Sybase
teradata	pydal.adapters.teradata.Teradata
vertica	pydal.adapters.mssql.Vertica

the uri string is then parsed in more detail by the adapter itself. An updated list of adapters can be obtained as dictionary with

For any adapter you can replace the driver with a different one globally (not thread safe):

```
import MySQLdb as mysqldb
from pydal.adapters.mysql import SQLAdapter
SQLAdapter.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={})
```

For recognized adapters you can also simply specify the name in the `adapter_args`:

```
from pydal.adapters.mysql import MySQL
assert "mysqldv" in MySQL.drivers
db = DAL(..., driver_args={}, adapter_args={"driver": "mysqldb"})
```

7.15.2 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.

7.15.3 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 in the aftermath (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).

7.15.4 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 in the aftermath (by accessing the database using the Google dashboard and deleting any corrupted entry from the table called `py4web_filesystem`.

7.15.5 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)
```

7.15.6 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.

7.15.7 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.

The RestAPI

Since version 19.5.10 pyDAL includes a restful API [CIT0801] called RestAPI. It is inspired by GraphQL [CIT0802] and while it's not quite the same due to it being less powerful, it is in the spirit of py4web since it's more practical and easier to use.

Like GraphQL 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 GraphQL it allows the server to specify a policy and restrict which queries are allowed and which ones are not. They can be evaluated dynamically per request based on the user and the state of the server.

As the name implies RestAPI allows all standard methods: GET, POST, PUT, and DELETE. Each of them can be enabled or disabled based on the policy, for individual tables and individual fields.

Note Specifications might be subject to changes since this is a new feature.

In the examples below we assume a simple app called "superheroes":

```python
# in superheroes/__init__.py
import os
from py4web import action, request, Field, DAL
from pydal.restapi import RestAPI, Policy

# database definition
DB_FOLDER = os.path.join(os.path.dirname(__file__), 'databases')
if not os.path.isdir(DB_FOLDER):
    os.mkdir(DB_FOLDER)
db = DAL('sqlite://storage.sqlite', folder=DB_FOLDER)
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'))

# add example entries in db
```

[CIT0801] https://en.wikipedia.org/wiki/Representational_state_transfer
[CIT0802] https://graphql.org/

```python
if not db(db.person).count():
    db.person.insert(name='Clark Kent', job='Journalist')
    db.person.insert(name='Peter Park', job='Photographer')
    db.person.insert(name='Bruce Wayne', job='CEO')
    db.superhero.insert(name='Superman', real_identity=1)
    db.superhero.insert(name='Spiderman', real_identity=2)
    db.superhero.insert(name='Batman', real_identity=3)
    db.superpower.insert(description='Flight')
    db.superpower.insert(description='Strength')
    db.superpower.insert(description='Speed')
    db.superpower.insert(description='Durability')
    db.tag.insert(superhero=1, superpower=1, strength=100)
    db.tag.insert(superhero=1, superpower=2, strength=100)
    db.tag.insert(superhero=1, superpower=3, strength=100)
    db.tag.insert(superhero=1, superpower=4, strength=100)
    db.tag.insert(superhero=2, superpower=2, strength=50)
    db.tag.insert(superhero=2, superpower=3, strength=75)
    db.tag.insert(superhero=2, superpower=4, strength=10)
    db.tag.insert(superhero=3, superpower=2, strength=80)
    db.tag.insert(superhero=3, superpower=3, strength=20)
    db.tag.insert(superhero=3, superpower=4, strength=70)
    db.commit()

# policy definitions
policy = Policy()
policy.set('superhero', 'GET', authorize=True, allowed_patterns=['*'])
policy.set('*', 'GET', authorize=True, allowed_patterns=['*'])

# for security reasons we disabled here all methods but GET at the policy level,
# to enable any of them just set authorize = True
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'])
@action.uses(db)
def api(tablename, rec_id=None):
    return RestAPI(db, policy)(request.method,
                              tablename,
                              rec_id,
                              request.GET,
                              request.POST
                              )

@action("index")
def index():
    return "RestAPI example"
```

8.1 RestAPI policies and actions

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/api/{tablename}
/superheroes/api/{tablename}/{rec_id}
```

The result can be seen directly with a browser, rendered as JSON. Let's look for example at the `person` table:

The diagram of the superhero's database should help you interpreting the code:

Note Keep in mind that **request.POST** only contains the form data that is posted using a **regular HTML-form** or **JavaScript FormData** object.

8.2 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 being queried as in:

    All superheroes called "Superman"

    ```
    /superheroes/api/superhero?name.eq=Superman
    ```

- the name of a field of a table referred by the table being queried as in:

    All superheroes with real identity "Clark Kent"

    ```
    /superheroes/api/superhero?real_identity.name.eq=Clark Kent
    ```

- the name of a field of a table that refers to the table being queried as in:

    All superheroes with any tag superpower with strength > 90

    ```
    /superheroes/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)

- a field of the table referenced by a many-to-many linked table as in:

    All superheroes with the flight power

```
/superheroes/api/superhero?superhero.tag.superpower.description.eq=Flight
```

Hint The key to understand the syntax above is to 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** equal 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.

8.3 RestAPI practical examples

Here are some practical examples:

URL:

```
/superheroes/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/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": "2021-01-04T07:03:38.466030",
    "model": [
        {
            "regex": "[1-9]\\d*",
            "name": "id",
            "default": null,
            "required": false,
            "label": "Id",
            "post_writable": true,
            "referenced_by": [
                "tag.superhero"
            ],
            "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/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/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/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": "2021-01-04T07:03:38.559918",
    "api_version": "0.1"
```

```
}
```

URL:

```
/superheroes/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/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 (it's a single line, split for readability):

```
/superheroes/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 (it's a single line, split for readability):

```
/superheroes/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
        },
        {
```

```json
        "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 (it's a single line, split for readability):

```
/superheroes/api/superhero?
@lookup=powers!:superhero.tag[strength].superpower[description],
identity!:real_identity[name]
```

OUTPUT:

```json
{
    "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": "2021-01-04T07:31:34.974953",
    "api_version": "0.1"
}
```

URL:

```
/superheroes/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/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/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/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"
}
```

8.4 The RestAPI response

All RestAPI response have the fields:

api_version RestAPI version.

timestamp Datetime in ISO 8601 format.

status RestAPI status (i.e. "success" or "error").

code HTTP status.

Other optional fields are:

count Total matching (not total returned), for GET.

items In response to a GET.

errors Usually a validation error.

models Usually if status != "success".

message For error details.

YATL Template Language

py4web uses two distinct template languages for rendering dynamic HTML pages that contain Python code:

- yatl (Yet Another Template Language) , which is considered the original reference implementation

- Renoir, which is a newer and faster implementation of yatl with additional functionality

Since `Renoir` does not include HTML helpers (see next chapter), py4web by default uses the `Renoir` module for rendering templates and the `yatl` module for helpers, plus some minor trickery to make them work together seamlessly.

py4web also uses double square brackets `[[ ... ]]` to escape Python code embedded in HTML, unless specified otherwise.

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 templates.

> **Warning** Be careful not to mix Python code square brackets with other square brackets! For example, you'll soon see syntax like this:
>
> ```
> [[items = ['a', 'b', 'c']]] # this gives "Internal Server Error"
> [[items = ['a', 'b', 'c'] ]] # this works
> ```
>
> It's mandatory to add a space after the first closed bracket for separating the list from the Python code square brackets.

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 un-indented Python inside the `[[ ... ]]` tags. But 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 template, the error report shows the generated template code, not the actual template 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` (see Section 10.2.1 later for details).

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.

Note While the response object used inside the controllers is a full `bottle.response` object, inside the yatl templates it is replaced by a dummy object (`yatl.template.DummyResponse`). This object is quite different, and much simpler: it only has a write method! Also, you never need to (and never should) call the `response.write` method explicitly.

9.1 Basic syntax

The py4web template language supports all Python control structures. Here we provide some exam-

---

plesof each of them. They can be nested according to usual programming practice. You can easily test them by copying the _scaffold app (see Section 5.5) and then editing the file `new_app/template/index.html`.

9.1.1 `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.

9.1.2 `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>
```

9.1.3 `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:

9.1. Basic syntax 137

```
[[
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>
```

9.1.4 `try...except...else...finally`

It is also possible to use `try...except` statements in templates 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.

9.1.5 `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 template must terminate with a `return` statement, or the automatic indentation will fail.

9.2 Information workflow

For dynamically modifying the workflow of the information there are custom commands available: `extend`, `include`, `block` and `super`. Note that they are special template directives, not Python commands.

In addition, you can use normal Python functions inside templates.

9.2.1 `extend` and `include`

Templates can extend and include other templates in a tree-like structure.

For example, we can think of a template "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 template**. Just like any other HTML template file, you can edit it from the command line or using the py4web Dashboard. The file name "layout.html" is just a convention.

Here is a minimalist page that extends the "layout.html" template and includes the "page.html" template:

```
<!--minimalist_page.html-->
[[extend 'layout.html']]
<h1>Hello World</h1>
[[include 'page.html']]
```

The extended layout file must contain an `[[include]]` directive, something like:

```
<!--layout.html-->
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    [[include]]
  </body>
</html>
```

When the template is called, the extended (layout) template is loaded, and the calling template replaces the `[[include]]` directive inside the layout. If you don't write the `[[include]]` directive inside the layout, then it will be included at the beginning of the file. Also, if you use multiple `[[extend]]` directives only the last one will be processed. 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 orig-

inaltemplate files themselves. So, the bytecode compiled version of a given template is a single .pyc file that includes the Python code not just for the original template file, but for its entire tree of extended and included templates.

Any content or code that **precedes** the `[[extend ...]]` directive will be inserted (and therefore executed) before the beginning of the extended template's content/code. Although this is not typically used to insert actual HTML content before the extended template's content, it can be useful as a means to define variables or functions that you want to make available to the extended template. For example, consider a template "index.html":

```
<!--index.html-->
[[sidebar_enabled=True]]
[[extend 'layout.html']]
<h1>Home Page</h1>
```

and an excerpt from "layout.html":

```
<!--layout.html-->
[[include]]
[[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.

It is also worth pointing out that the variables returned by the controller function are available not only in the function's main template, but in all of its extended and included templates as well.

9.2.2 Extending using variables

The argument of an `extend` or `include` (i.e., the extended or included template name) can be a Python variable (though not a Python expression). However, this imposes a limitation – templates that use variables in `extend` or `include` statements cannot be bytecode compiled. As noted above, bytecode-compiled templates include the entire tree of extended and included templates, so the specific extended and included templates must be known at compile time, which is not possible if the template names are variables (whose values are not determined until run time). Because bytecode compiling templates 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_template.html']]
[[else:]]
    [[include 'that_template.html']]
[[pass]]
```

The above code does not present any problem for bytecode compilation because no variables are involved. Note, however, that the bytecode compiled template will actually include the Python code for both "this_template.html" and "that_template.html", though only the code for one of those templates 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.

9.2.3 Template Functions

Consider this "layout.html":

```
<!--layout.html-->
<html>
  <body>
    [[include]]
    <div class="sidebar">
      [[if 'mysidebar' in globals():]][[mysidebar()]][[else:]]
        my default sidebar
      [[pass]]
    </div>
  </body>
</html>
```

and this extending template

```
[[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 template 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 template function using `[[mysidebar()]]` rather than `[[=mysidebar()]]`. Functions defined in this way can take arguments.

9.2.4 `block` and `super`

The main way to make a template 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 template layout.html. This template is extended by the template `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 simplified 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 template `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 template, yet the enclosing DIV and class comes from layout.html. This allows consistency across templates:

```
<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 template but not in the extending template, the content of the extended template 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 template.

Inside a block, you can use the expression `[[super]]` to include the content of the parent. For example, if we replace the above extending template 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>
```

9.3 Page layout standard structure

9.3.1 Default page layout

The "templates/layout.html" that currently ships with the py4web _scaffold_ application is quite complex but it has the following structure:

```
<!DOCTYPE html>
<html>
  <head>
    <base href="[[=URL('static')]]/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="shortcut icon"
href="data:image/x-icon;base64,AAABAAEAAQEAAAEAIAAwAAAAFgAAACgAAAABAAAAAgAAAAEAIAAAAAAABAAAAAAA
    <link rel="stylesheet" href="css/no.css">
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.14.0/css/all.min.css"
integrity="sha512-1PKOgIY59xJ8Co8+NE6FZ+LOAZKjy+KY8iq0G4B3CyeY6wYHN3yt9PW0XpSriVlkMXe40PTKnXrLnZ
crossorigin="anonymous" />
    <style>.py4web-validation-error{margin-top:-16px;
font-size:0.8em;color:red}</style>
    [[block page_head]]<!-- individual pages can customize header here -->[[end]]
  </head>
  <body>
    <header>
      <!-- Navigation bar -->
      <nav class="black">
        <!-- Logo -->
        <a href="[[=URL('index')]]">
          <b>py4web
<script>document.write(window.location.href.split('/')[3]);</script></b>
        </a>
        <!-- Do not touch this -->
        <label for="hamburger">?</label>
        <input type="checkbox" id="hamburger">
        <!-- Left menu ul/li -->
        [[block page_left_menu]][[end]]
        <!-- Right menu ul/li -->
        <ul>
          [[if globals().get('user'):]]
          <li>
            <a class="navbar-link is-primary">
              [[=globals().get('user',{}).get('email')]]
            </a>
            <ul>
              <li><a href="[[=URL('auth/profile')]]">Edit Profile</a></li>
              <li><a href="[[=URL('auth/change_password')]]">Change
Password</a></li>
              <li><a href="[[=URL('auth/logout')]]">Logout</a></li>
            </ul>
          </li>
          [[else:]]
          <li>
            Login
            <ul>
              <li><a href="[[=URL('auth/register')]]">Sign up</a></li>
              <li><a href="[[=URL('auth/login')]]">Log in</a></li>
            </ul>
          </li>
```

```
            [[pass]]
          </ul>
        </nav>
      </header>
      <!-- beginning of HTML inserted by extending template -->
      <center>
        <div>
          <!-- Flash alert messages, first optional one in data-alert -->
          <flash-alerts class="padded"
 data-alert="[[=globals().get('flash','')]]"></flash-alerts>
        </div>
        <main class="padded">
          <!-- contect injected by extending page -->
          [[include]]
        </main>
      </center>
      <!-- end of HTML inserted by extending template -->
      <footer class="black padded">
        <p>
          Made with py4web
        </p>
      </footer>
    </body>
    <!-- You've gotta have utils.js -->
    <script src="js/utils.js"></script>
    [[block page_scripts]]<!-- individual pages can add scripts here -->[[end]]
  </html>
```

There are a few features of this default layout that make it very easy to use and customize:

- it is written in HTML5

- on line 7 it's used the `no.css` stylesheet, see here

- on line 58 `[[include]]` is replaced by the content of the extending template when the page is rendered

- it contains the following blocks: page_head, page_left_menu, page_scripts

- on line 30 it checks if the user is logged on and changes the menu accordingly

- on line 54 it checks for flash alert messages

Of course you can also completely replace the "layout.html" and the stylesheet with your own.

9.3.2 Mobile development

Although the default layout.html is designed to be mobile-friendly, one may sometimes need to use different templates when a page is visited by a mobile device.

YATL helpers

10.1 Helpers overview

Consider the following code in a template:

```
[[=DIV('this', 'is', 'a', 'test', _id='123', _class='myclass')]]
```

it is rendered as:

```
<div id="123" class="myclass">thisisatest</div>
```

You can easily test the rendering of these commands by copying the _scaffold app (see Section 5.5) and then editing the file new_app/template/index.html.

DIV is a **helper class**, i.e., something that can be used to build HTML programmatically. It corresponds to the HTML <div> tag.

Helpers can have:

- **positional arguments** interpreted as objects contained between the open and close tags, like thisisatest in the previous example
- **named arguments** (start with an underscore) interpreted as HTML tag attributes (without the underscore), like _class and _id in the previous example
- **named arguments** (start without an underscore), in this case 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 are the current set of helpers available within the YATL module:

A, BEAUTIFY, BODY, CAT, CODE, DIV, EM, FORM, H1, H2, H3, H4, H5, H6, HEAD, HTML, IMG, INPUT, LABEL, LI, METATAG, OL, OPTION, P, PRE, SELECT, SPAN, STRONG, TABLE, TAG, TAGGER, THEAD, TBODY, TD, TEXTAREA, TH, TT, TR, UL, XML, xmlescape, I, META, LINK, TITLE, STYLE, SCRIPT

Helpers can be used to build complex expressions, that can then be serialized to XML. For example:

```
[[=DIV(STRONG(I("hello ", "<world>")), _class="myclass")]]
```

is rendered:

```
<div class="myclass"><strong><i>hello &lt;world&gt;</i></strong></div>
```

Helpers can also be serialized into strings, equivalently, with the __str__ and the xml methods. This can be manually tested directly with a Python shell or by using the Section 3.6.6 of py4web and then:

```
>>> from yatl.helpers import *
>>>
>>> str(DIV("hello world"))
'<div>hello world</div>'
>>> 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(STRONG('x'))
>>> a[0][0] = 'y'
>>> print(a)
<div><span>yb</span><strong>x</strong></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.children, and the complete set of attributes can be accessed via a dictionary called a.attributes. So, a[i] is equivalent to a.children[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 can 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>
```

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>
```

10.2 Built-in helpers

10.2.1 `XML`

`XML` is an helper 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("<strong>hello</strong>"))
<div>&lt;strong&gt;hello&lt;/strong&gt;</div>
```

by using `XML` you can prevent escaping:

```
>>> print(DIV(XML("<strong>hello</strong>")))
<div><strong>hello</strong></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. In this case 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/',
        'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'table', 'tr', 'td',
        'div', 'strong', 'span'],
    allowed_attributes={'a': ['href', 'title', 'target'],
        'img': ['src', 'alt'], 'blockquote': ['type'], 'td': ['colspan']})
```

10.2.2 `A`

This helper is used to build links.

```
>>> print(A('<click>', XML('<strong>me</strong>'),
            _href='http://www.py4web.com'))
<a href="http://www.py4web.com">&lt;click&gt;<strong>me</strong></a>
```

10.2.3 `BODY`

This helper makes the body of a page.

```
>>> print(BODY('<hello>', XML('<strong>world</strong>'), _bgcolor='red'))
<body bgcolor="red">&lt;hello&gt;<strong>world</strong></body>
```

10.2.4 `CAT`

This helper concatenates other helpers.

```
>>> print(CAT('Here is a ', A('link', _href='target'), ', and here is some ',
STRONG('bold text'), '.'))
Here is a <a href="target">link</a>, and here is some <strong>bold text</strong>.
```

10.2.5 `DIV`

This is the content division element.

```
>>> print(DIV('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<div id="0" class="test">&lt;hello&gt;<strong>world</strong></div>
```

10.2.6 `EM`

Emphasizes its content.

```
>>> print(EM('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<em id="0" class="test">&lt;hello&gt;<strong>world</strong></em>
```

10.2.7 `FORM`

Use this helper to make a FORM for user input. Forms will be later discussed in detail in the dedicated Chapter 12 chapter.

```
>>> print(FORM(INPUT(_type='submit'), _action='', _method='post'))
<form action="" method="post"><input type="submit"/></form>
```

10.2.8 `H1, H2, H3, H4, H5, H6`

These helpers are for paragraph headings and subheadings.

```
>>> print(H1('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<h1 id="0" class="test">&lt;hello&gt;<strong>world</strong></h1>
```

10.2.9 `HEAD`

For tagging the HEAD of an HTML page.

```
>>> print(HEAD(TITLE('<hello>', XML('<strong>world</strong>'))))
<head><title>&lt;hello&gt;<strong>world</strong></title></head>
```

10.2.10 `HTML`

For tagging an HTML page.

```
>>> print(HTML(BODY('<hello>', XML('<strong>world</strong>'))))
<html><body>&lt;hello&gt;<strong>world</strong></body></html>
```

10.2.11 `I`

This helper makes its contents italic.

```
>>> print(I('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<i id="0" class="test">&lt;hello&gt;<strong>world</strong></i>
```

10.2.12 `IMG`

It can be used to embed images into HTML.

```
>>> print(IMG(_src='http://example.com/image.png', _alt='test'))
<img alt="test" src="http://example.com/image.png"/>
```

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="/default/index"><img alt="My Logo" src="/static/logo.png"/></a>
```

10.2.13 `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 name="test" value="a"/>
```

For radio buttons use the `_checked` attribute:

```
>>> for v in ['a', 'b', 'c']:
...     print(INPUT(_type='radio', _name='test', _value=v, _checked=v=='b'), v)
...
<input name="test" type="radio" value="a"/> a
<input checked="checked" name="test" type="radio" value="b"/> b
<input name="test" type="radio" value="c"/> c
```

and similarly for checkboxes:

```
>>> print(INPUT(_type='checkbox', _name='test', _value='a', _checked=True))
<input checked="checked" name="test" type="checkbox" value="a"/>
>>> print(INPUT(_type='checkbox', _name='test', _value='a', _checked=False))
<input name="test" type="checkbox" value="a"/>
```

10.2.14 `LABEL`

It is used to create a LABEL tag for an INPUT field.

```
>>> print(LABEL('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<label id="0" class="test">&lt;hello&gt;<strong>world</strong></label>
```

10.2.15 `LI`

It makes a list item and should be contained in a `UL` or `OL` tag.

```
>>> print(LI('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<li id="0" class="test">&lt;hello&gt;<strong>world</strong></li>
```

10.2.16 `OL`

It stands for ordered list. The list should contain LI tags.

```
>>> print(OL(LI('<hello>'), LI(XML('<strong>world</strong>')), _class='test',
_id=0))
<ol class="test" id="0"><li>&lt;hello&gt;</li><li><strong>world</strong></li></ol>
```

10.2.17 `OPTION`

This should only be used as argument of a `SELECT`.

```
>>> print(OPTION('<hello>', XML('<strong>world</strong>'), _value='a'))
<option value="a">&lt;hello&gt;<strong>world</strong></option>
```

For selected options use the `_selected` attribute:

```
>>> print(OPTION('Thank You', _value='ok', _selected=True))
<option selected="selected" value="ok">Thank You</option>
```

10.2.18 `P`

This is for tagging a paragraph.

```
>>> print(P('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<p id="0" class="test">&lt;hello&gt;<strong>world</strong></p>
```

10.2.19 `PRE`

Generates a `<pre>...</pre>` tag for displaying pre-formatted text. The `CODE` helper is generally preferable for code listings.

```
>>> print(SELECT(OPTION('first', _value='1'), OPTION('second', _value='2'),
_class='test', _id=0))
<pre id="0" class="test">&lt;hello&gt;<strong>world</strong></pre>
```

10.2.20 `SCRIPT`

This is for include or link a script, such as JavaScript.

```
>>> print(SCRIPT('console.log("hello world");', _type='text/javascript'))
<script type="text/javascript">console.log("hello world");</script>
```

10.2.21 `SELECT`

Makes a `<select>...</select>` tag. This is used with the `OPTION` helper.

```
>>> print(SELECT(OPTION('first', _value='1'), OPTION('second', _value='2'),
... _class='test', _id=0))
<select class="test" id="0"><option value="1">first</option><option
value="2">second</option></select>
```

10.2.22 `SPAN`

Similar to `DIV` but used to tag inline (rather than block) content.

```
>>> print(SPAN('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<span id="0" class="test">&lt;hello&gt;<strong>world</strong></span>
```

10.2.23 `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>
```

10.2.24 **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.

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(*map(TD, table[0])), TR(*map(TD, 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(*map(TD, rows)) for rows in table]))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

10.2.25 **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(TD('<hello>')), _class='test', _id=0))
<tbody id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tbody>
```

10.2.26 **TEXTAREA**

This helper makes a `<textarea>...</textarea>` tag.

```
>>> print(TEXTAREA('<hello>', XML('<strong>world</strong>'), _class='test',
... _cols="40", _rows="10"))
<textarea class="test" cols="40"
rows="10">&lt;hello&gt;<strong>world</strong></textarea>
```

10.2.27 **TH**

This is used instead of `TD` in table headers.

```
>>> print(TH('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<th id="0" class="test">&lt;hello&gt;<strong>world</strong></th>
```

10.2.28 **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>
```

10.2.29 `TITLE`

This is used to tag the title of a page in an HTML header.

```
>>> print(TITLE('<hello>', XML('<strong>world</strong>')))
<title>&lt;hello&gt;<strong>world</strong></title>
```

10.2.30 `TT`

Tags text as typewriter (monospaced) text.

```
>>> print(TT('<hello>', XML('<strong>world</strong>'), _class='test', _id=0))
<tt id="0" class="test">&lt;hello&gt;<strong>world</strong></tt>
```

10.2.31 `UL`

It stands for unordered list. The list should contain LI tags.

```
>>> print(UL(LI('<hello>'), LI(XML('<strong>world</strong>')), _class='test',
_id=0))
<ul class="test" id="0"><li>&lt;hello&gt;</li><li><strong>world</strong></li></ul>
```

10.2.32 `URL`

The URL helper is not part of yatl package, instead it is provided by py4web.

10.3 Custom helpers

10.3.1 `TAG`

Sometimes you need to generate **custom XML tags***. For this purpose 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')]]
```

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 an helper instance.

10.3.2 `BEAUTIFY`

`BEAUTIFY` is used to build HTML representations of compound objects, including lists, tuples and dictionaries:

```
[[=BEAUTIFY({"a": ["hello", STRONG("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", STRONG("world")], "b": (1, 2)}
```

will render as:

```html
<table><tbody>
<tr><th>a</th><td><ul><li>hello</li><li><strong>world</strong></li></ul></td></tr>
<tr><th>b</th><td>(1, 2)</td></tr>
</tbody></table>
```

10.4 Server-side *DOM*

As we've already seen the helpers mechanism in py4web also provides a server-side representation of the document object model (DOM).

10.4.1 `children`

Each helper object keep the list of its components into the `children` attribute.

```
>>> CAT('hello', STRONG('world')).children
['hello', <yatl.helpers.TAGGER object at 0x7fa533ff7640>]
```

10.4.2 `find`

To help searching into the DOM, all helpers have a `find` method with the following signature:

```python
def find(self, query=None, **kargs)
```

that returns all the components matching supplied arguments.

A very simple `query` can be a tag name:

```python
>>> a = DIV(DIV(SPAN('x'), 3, DIV(SPAN('y'))))
>>> for c in a.find('span', first_only=True): c[0]='z'
>>> print(a)  # We should .xml() here instead of print
<div><div><span>z</span>3<div><span>y</span></div></div></div>
>>> for c in a.find('span'): c[0]='z'
>>> print(a)
<div><div><span>z</span>3<div><span>z</span></div></div></div>
```

It also supports a syntax compatible with jQuery, accepting the following expressions:

- jQuery Multiple Selector, e.g. "selector1, selector2, selectorN",
- jQuery Descendant Selector, e.g. "ancestor descendant",
- jQuery ID Selector, e.g. "#id",
- jQuery Class Selector, e.g. ".class", and
- jQuery Attribute Equals Selector, e.g. "[name=value]", notice that here the value must be unquoted.

Here are some examples:

```
>>> a = DIV(SPAN(A('hello', **{'_id': '1-1', '_u:v': '$'})), P('world',
_class='this is a test'))
>>> for e in a.find('div a#1-1, p.is'): print(e)
<a id="1-1" u:v="$">hello</a>
<p class="this is a test">world</p>
>>> for e in a.find('#1-1'): print(e)
<a id="1-1" u:v="$">hello</a>
>>> a.find('a[u:v=$]')[0].xml()
'<a id="1-1" u:v="$">hello</a>'
>>> a = FORM(INPUT(_type='text'), SELECT(OPTION(0)), TEXTAREA())
>>> for c in a.find('input, select, textarea'): c['_disabled'] = True
>>> a.xml()
'<form><input disabled="disabled" type="text"/><select
disabled="disabled"><option>0</option></select><textarea
disabled="disabled"></textarea></form>'
>>> for c in a.find('input, select, textarea'): c['_disabled'] = False
>>> a.xml()
'<form><input
type="text"/><select><option>0</option></select><textarea></textarea></form>'
```

Elements that are matched can also be replaced or removed by specifying a `replace` argument (note, a list of the original matching elements is still returned as usual).

```
>>> a = DIV(DIV(SPAN('x', _class='abc'), DIV(SPAN('y', _class='abc'), SPAN('z',
_class='abc'))))
>>> b = a.find('span.abc', replace=P('x', _class='xyz'))
>>> print(a)
<div><div><p class="xyz">x</p><div><p class="xyz">x</p><p
class="xyz">x</p></div></div></div>
```

`replace` can be a callable, which will be passed the original element and should return a new element to replace it.

```
>>> a = DIV(DIV(SPAN('x', _class='abc'), DIV(SPAN('y', _class='abc'), SPAN('z',
_class='abc'))))
>>> b = a.find('span.abc', replace=lambda el: P(el[0], _class='xyz'))
>>> print(a)
<div><div><p class="xyz">x</p><div><p class="xyz">y</p><p
class="xyz">z</p></div></div></div>
```

If `replace=None`, matching elements will be removed completely.

```
>>> a = DIV(DIV(SPAN('x', _class='abc'), DIV(SPAN('y', _class='abc'), SPAN('z',
_class='abc'))))
>>> b = a.find('span', text='y', replace=None)
>>> print(a)
<div><div><span class="abc">x</span><div><span class="abc"></span><span
class="abc">z</span></div></div></div>
```

If a `text` argument is specified, elements will be searched for text components that match text, and any matching text components will be replaced (`text` is ignored if `replace` is not also specified, use a `find` argument when you only need searching for textual elements).

Like the `find` argument, `text` can be a string or a compiled regex.

```
>>> a = DIV(DIV(SPAN('x', _class='abc'), DIV(SPAN('y', _class='abc'), SPAN('z',
_class='abc'))))
>>> b = a.find(text=re.compile('x|y|z'), replace='hello')
>>> print(a)
<div><div><span class="abc">hello</span><div><span class="abc">hello</span><span
class="abc">hello</span></div></div></div>
```

If other attributes are specified along with `text`, then only components that match the specified attributes will be searched for text.

```
>>> a = DIV(DIV(SPAN('x', _class='abc'), DIV(SPAN('y', _class='efg'), SPAN('z',
_class='abc'))))
>>> b = a.find('span.efg', text=re.compile('x|y|z'), replace='hello')
>>> print(a)
<div><div><span class="abc">x</span><div><span class="efg">hello</span><span
class="abc">z</span></div></div></div>
```

10.5 Using Inject

Normally all the code should be called from the controller program, and only the necessary data is passed to the template in order to be displayed. But sometimes it's useful to pass variables or even use a python function as a helper called from a template.

In this case you can use the fixture `Inject` from py4web.utils.factories.

This is a simple example for injecting a variable:

```python
from py4web.utils.factories import Inject

my_var = "Example variable to be passed to a Template"


...


@unauthenticated("index", "index.html")
@action.uses(Inject(my_var=my_var))
def index():

    ...
```

Then in `index.html` you can use the injected variable:

```
[[=my_var]]
```

You can also use `Inject` to add variables to the auth.enable line; in this way auth forms would have access to that variable.

```
auth.enable(uses=(session, T, db, Inject(TIMEOFFSET=settings.TIMEOFFSET)))
```

A more complex usage of Inject is for passing python functions to templates. For example if your helper function is called *sidebar_menu* and it's inside the libs/helpers.py module of your app, you could use this in **controllers.py**:

```python
from py4web.utils.factories import Inject
from .libs.helpers import sidebar_menu

@action(...)
@action.uses("index.html", Inject(sidebar_menu=sidebar_menu))
def index(): ....
```

OR

```python
from py4web.utils.factories import Inject
from .libs import helpers

@action(...)
@action.uses(Inject(**vars(helpers)), "index.html")
def index(): ....
```

Then you can import the needed code in the index.html template in a clean way:

```
[[=sidebar_menu]]
```

Internationalization

11.1 Pluralize

Pluralize is a Python library for Internationalization (i18n) and Pluralization (p10n).

The library assumes a folder (for example "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.

11.2 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')
```

Forms

The Form class provides a high-level API for quickly building CRUD (create, update and delete) forms, especially for working on an existing database table. It can generate and process a form from a list of desired fields and/or from an existing database table.

There are 3 types of forms:

CRUD Create forms:

```
@action('create_thing')
@action.uses('generic.html', db, flash)
def create_thing():
    form = Form(db.thing)
    if form.accepted:
        flash.set("record created")
        redirect(URL('other_page'))
    return locals()
```

CRUD Update forms:

```
@action('update_thing/<thing_id:int>')
@action.uses('generic.html', db, flash)
def update_thing(thing_id):
    form = Form(db.thing, thing_id)
    if form.accepted:
        flash.set("record updated")
        redirect(URL('other_page'))
    return locals()
```

Non-CRUD forms (not associated to a database):

```
@action('some_form')
@action.uses('generic.html', flash)
def some_form():
    fields = [
        Field("name", requires=IS_NOT_EMPTY()),
        Field("color", required=IS_IN_SET(["red","blue","green"])),
    ]
    form = Form(fields)
    if form.accepted:
        flash.set("information recorded")
        redirect(URL('other_page'))
    return locals()
```

The use of flash is optional. `flash` is defined in `common.py` in the scaffolding application. It simply stores a message in a cookie so it can be recovered and displayed after redirection. This is done in the default layout.

In this chapter from now on we will assume the following model and an app derived from the scaffolding app:

```
db.define_table(
    'thing',
    Field('name', requires=IS_NOT_EMPTY()),
    Field('color', requires=IS_IN_SET(['red','blue','green'])),
    Field('image', 'upload', download_url=lambda name: URL('download', name)),
)
```

12.1 The Form constructor

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,
    validation=None,
    csrf_session=None,
    csrf_protection=True,
    lifespan=None,
    signing_info=None,
    ):
```

Where:

- `table`: a DAL table or a list of fields

- `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.
    Can be `FormStyleDefault` (default), `FormStyleBulma`, `FormStyleBootstrap4`, or `FormStyleBootstrap5`.

- `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

- `validation`: an optional validator, see Section 12.6.8

- `csrf_session`: if None, no csrf token is added. If a session, then a CSRF token is added and verified

- `lifespan`: lifespan of CSRF token in seconds, to limit form validity

- `signing_info`: information that should not change between when the CSRF token is signed and verified

`FormStyleDefault` is an object that is used by default for every form and it is defined in `py4web.utils.form.FormStyleDefault`. You should never change it but you can make a copy and pass it as `formstyle`.

You can change the style of a field named, for example, `color` by changing the attribute `color` of the FormStyle.

12.2 A minimal form example without a database

Let's start with a minimal working form example. Create a new minimal app called `form_minimal`:

```python
# in controllers.py
from py4web impot action, redirect, URL, Field
from py4web.utils.form import Form
from pydal.validators import *

@action('index', method=['GET', 'POST'])
@action.uses('form_minimal.html')
def index():
    fields = [
        Field('name', requires=IS_NOT_EMPTY()),
        Field('color', requires=IS_IN_SET(['red','blue','green'])),
    ]
    form = Form(fields)
    if form.accepted:
        # Do something with form.vars['name'] and form.vars['color']
        redirect(URL('accepted'))
    if form.errors:
        # do something
        ...
    return dict(form=form)

@action("accepted")
def accepted():
    return "form_example accepted"
```

Also, you need to create a file inside the app called `templates/form_minimal.html` that just contains the line:

```
[[extend 'layout.html']]
[[=form]]
```

Then reload py4web and visit http://127.0.0.1:8000/form_minimal - you'll get the Form page:

Name

Enter a value

Color

red ▼

Submit

Note that:

- `Form` is a class contained in the `py4web.utils.form` module

- it's possible to use **form validators** like `IS_NOT_EMPTY`, see Section 12.6 later. They are imported from the `pydal.validators` module

- it's normally important to use both the **GET** and the **POST** methods in the action where the form is contained

This example is intentionally not using a database, a template, nor the session management. The next example will.

12.3 Basic form example

In this next basic example we generate a CRUD create form from a database. Create a new minimal app called `form_basic`:

```python
# in controllers.py
from py4web import action, redirect, URL, Field
from py4web.utils.form import Form
from pydal.validators import *
from .common import db

# controllers definition
@action("create_form", method=["GET", "POST"])
@action.uses("form_basic.html", db)
def create_form():
    form = Form(db.thing)
    rows = db(db.thing).select()
    return dict(form=form, rows=rows)
```

Note the import of two simple validators on top, in order to be used later with the `requires` parameter. We'll fully explain them on the Section 12.6 paragraph.

You will also need a template file `templates/form_basic.html` that contains, for example, the following code:

```
[[extend "layout.html"]]

<h2 class="title">Form Basic example: My Things</h2>

[[=form]]

<h2 class="title">Rows</h2>

<ul>
[[for row in rows:]]
<li>[[=row.id]]: [[=row.name]] has color [[=row.color]]</li>
[[pass]]
</ul>
```

Reload py4web and visit http://127.0.0.1:8000/create_form : the result is an input form on the top of the page, and the list of all the previously added entries on the bottom:

Form Basic example: My Things

Name

Color

red

Image

Upload: Choose File No file chosen

Submit

Rows

12: Chair has color blue

This is a simple example and you cannot change nor delete existing records. But if you'd like to experiment, the database content can be fully seen and changed with the Dashboard app.

You can turn a create form into a CRUD update form by passing a record or a record id it second argument:

```
# controllers definition
```

```
@action("update_form/<thing_id:int>", method=["GET", "POST"])
@action.uses("form_basic.html", db)
def update_form(thing_id):
    form = Form(db.thing, thing_id)
    rows = db(db.thing).select()
    return dict(form=form, rows=rows)
```

12.3.1 File upload field

We can make a minor modification to our reference model and an upload type file:

```
db.define_table(
    'thing',
    Field('name', requires=IS_NOT_EMPTY()),
    Field('color', requires=IS_IN_SET(['red','blue','green'])),
    Field('image', 'upload', download_url=lambda image: URL('download', image)),
)
```

The file upload field is quite particular. The standard way to use it (as in the _scaffold app) is to have the UPLOAD_FOLDER defined in the common.py file. But if you don't specify it, then the default value of `your_app/upload` folder will be used (and the folder will also be created if needed). `download_url` is a callback that given the image name, generated the URL to download. The `download` url is predefined in `common.py`.

We can modify `form_basic.html` to display the uploaded images:

```
<h2 class="title">Form upload example: My Things</h2>

[[=form]]

<h2 class="title">Rows</h2>

<ul>
[[for row in rows:]]
<li>[[=row.id]]: [[=row.name]] has color [[=row.color]]
    <img src="[[=URL('download', row.image)]]" />
[[pass]]
</ul>
```

The uploaded files (the thing images) are saved on the UPLOAD_FOLDER folder with their name hashed. Other details on the upload fields can be found on Section 7.5 paragraph, including a way to save the files inside the database itself.

12.4 Widgets

12.4.1 Standard widgets

Py4web provides many widgets in the py4web.utility.form library. They are simple plugins that easily allow you to specify the type of the input elements in a form, along with some of their properties.

Here is the full list:

- CheckboxWidget
- DateTimeWidget
- FileUploadWidget
- ListWidget
- PasswordWidget
- RadioWidget

- SelectWidget

- TextareaWidget

This is an improved 'Basic Form Example' with a radio button widget:

```python
# in controllers.py
from py4web import action, redirect, URL, Field
from py4web.utils.form import Form, FormStyleDefault, RadioWidget
from pydal.validators import *
from .common import db

# controllers definition
@action("create_form", method=["GET", "POST"])
@action.uses("form_widgets.html", db)
def create_form():
    FormStyleDefault.widgets['color']=RadioWidget()
    form = Form(db.thing, formstyle=FormStyleDefault)
    rows = db(db.thing).select()
    return dict(form=form, rows=rows)
```

Notice the differences from the 'Basic Form example' we've seen at the beginning of the chapter:

- you need to import the widget from the py4web.utils.form library

- before the form definition, you define the `color` field form style with the line:

```
FormStyleDefault.widgets['color']=RadioWidget()
```

The result is the same as before, but now we have a radio button widget instead of the dropdown menu!

Using widgets in forms is quite easy, and they'll let you have more control on its pieces.

Important When using py4web, use py4web widgets, and do not use the pydal widget argument in the Field object (see Section 7.5).

12.4.2 Custom widgets

You can also customize the widgets properties by cloning and modifying and existing style. Let's have a quick look, improving again our Superhero example:

```python
# in controllers.py
from py4web import action, redirect, URL, Field
from py4web.utils.form import Form, FormStyleDefault, RadioWidget
from pydal.validators import *
from .common import db

# custom widget class definition
class MyCustomWidget:
    def make(self, field, value, error, title, placeholder, readonly=False):
        tablename = field._table if "_table" in dir(field) else "no_table"
        control = INPUT(
            _type="text",
            _id="%s_%s" % (tablename, field.name),
            _name=field.name,
            _value=value,
            _class="input",
            _placeholder=placeholder if placeholder and placeholder != "" else
"..",
            _title=title,
            _style="font-size: x-large;color: red; background-color: black;",
        )
```

```
        return control

# controllers definition
@action("create_form", method=["GET", "POST"])
@action.uses("form_custom_widgets.html", db)
def create_form():
    MyStyle = FormStyleDefault.clone()
    MyStyle.classes = FormStyleDefault.classes
    MyStyle.widgets['name']=MyCustomWidget()
    MyStyle.widgets['color']=RadioWidget()

    form = Form(db.thing, deletable=False, formstyle=MyStyle)
    rows = db(db.thing).select()
    return dict(form=form, rows=rows)
```

The result is similar to the previous ones, but now we have a custom input field, with foreground color red and background color black,

Even the radio button widget has changed, from red to blue.

12.5 Advanced form design

12.5.1 Form structure manipulation

In py4web a form is rendered by YATL helpers. This means the tree structure of a form can be manipulated before the form is serialized in HTML. Here is an example of how to manipulate the generate HTML structure:

```
db.define_table('paint', Field('color'))
form = Form(db.paint)
form.structure.find('[name=color]')[0]['_class'] = 'my-class'
```

Notice that a form does not make an HTML tree until form structure is accessed. Once accessed you can use .find(...) to find matching elements. The argument of find is a string following the filter syntax of jQuery. In the above case there is a single match [0] and we modify the _class attribute of that element. Attribute names of HTML elements must be preceded by an underscore.

12.5.2 Custom forms

Custom forms allow you to granulary control how the form is processed. In the template file, you can execute specific instructions before the form is displayed or after its data submission by inserting code among the following statements:

```
[[=form.custom.begin ]]
[[=form.custom.submit ]]
[[=form.custom.end ]]
```

For example you could use it to avoid displaying the id field while editing a record in your form:

```
[[extend 'layout.html']]
[[=form.custom.begin ]]
    [[for field in DETAIL_FIELDS: ]]
        [[ if field not in ['id']: ]]
            <div class="select">
                [[=form.custom.widgets[field] ]]
            </div>
        [[pass]]
    [[pass]]
[[=form.custom.submit ]]
[[=form.custom.end ]]
```

Note: 'custom' is just a convention, it could be any name that does not clash with already defined objects.

> **Warning** When working with custom forms, if you have a writable field that isn't included on your form, it will be set to null when you save a record. Any time a field is not included on a custom form, it should be set field.writable=False to ensure that field is not updated.
>
> Also, custom forms only create the element for a given field, but no surrounding elements that might be needed based on your css framework. For example, if you're using Bulma as your css framework, you'll have to add an outer DIV in order to get select controls to appear correctly.

You can also be more creative and use your HTML in the template instead of using widgets:

```
[[extend 'layout.html']]

[[for field, error form.errors.items:]]
<div class="error">Field [[=field]] [[=error]]</div>
[[pass]]

[[=form.custom.begin ]]

<div class="select">
    <input name="name" value="form.vars.get('name', '')"/>
</div>
<div class="select">
[[for color in ['red', 'blue', 'green']:]]
    <label>[[=color]]</label>
    <input name="color" type="radio" value="[[=color]]"
            [[if form.vars.get('color') == color:]]checked[[pass]]
    />
[[pass]]
</div>
<input type="submit" value="Click me"/>
[[=form.custom.end ]]
```

12.5.3 The sidecar parameter

The sidecar is the stuff injected in the form along with the submit button.

For example, you can inject a simple `click me` button in your form with the following code:

```
form.param.sidecar = DIV(BUTTON("click me", _onclick="alert('doh!')"))
```

In particular, this is frequently used for adding a `Cancel` button, which is not provided by py4web:

```
attrs = {
"_onclick": "window.history.back(); return false;",
"_class": "button is-default",
}
form.param.sidecar.append(BUTTON("Cancel", **attrs))
```

12.6 Form validation

Validators are classes used to validate input fields (including forms generated from database tables). They are normally assigned using the `requires` attribute of a table `Field` object, as already shown on the Section 7.5 paragraph of the DAL chapter. Also, you can use advanced validators in order to create widgets such as drop-down menus, radio buttons and even lookups from other tables. Last but not least, you can even explicitly define a validation function.

Here is a simple example of how to require a validator for a table field:

```
db.define_table(
    'person',
    Field('name',requires=IS_NOT_EMPTY()),
    Field('job')
)
```

The validator is frequently written explicitly outside the table definition in this equivalent syntax:

```
db.define_table(
    'person',
    Field('name'),
    Field('job')
)
db.person.name.requires = IS_NOT_EMPTY()
```

A field can have a single validator or a list of multiple validators:

```
db.person.name.requires = [
    IS_NOT_EMPTY(),
    IS_NOT_IN_DB(db, 'person.name')]
```

Mind that the only validators that can be used with `list:` type fields are:

- `IS_IN_DB(..., multiple=True)`

- `IS_IN_SET(..., multiple=True)`

- `IS_NOT_EMPTY()`

- `IS_LIST_OF_EMAILS()`

- `IS_LIST_OF(...)`

The latter can be used to apply any validator to the individual items in the list. `multiple=(1, 1000)` requires a selection of between 1 and 1000 items. This enforces selection of at least one choice.

Built-in validators have constructors that take an `error_message` argument:

```
IS_NOT_EMPTY(error_message='cannot be empty!')
```

Notice the error message is usually fist option of the constructors and you can normally avoid to name it. Hence the following syntax is equivalent:

If you want to use internationalization like explained in a previous chapter you need to define your own messages and wrap the validator message in the T operator:

IS_NOT_EMPTY(error_message=T('cannot be empty!'))

```
IS_NOT_EMPTY('cannot be empty!')
```

Here is an example of a validator on a database table:

```
db.person.name.requires = IS_NOT_EMPTY(error_message=T('fill this!'))
```

where we have used the translation operator `T` to allow for internationalization. Notice that error messages are not translated by default unless you define them explicitly with `T`.

One can also call validators explicitly for a field:

```
db.person.name.validate(value)
```

which returns a tuple `(value, error)` and `error` is `None` if the value validates.

You can easily test most of the following validators directly using python only. For example:

```
>>> from pydal.validators import *
>>> IS_ALPHANUMERIC()('test')
```

```
('test', None)
>>> IS_ALPHANUMERIC()('test!')
('test!', 'Enter only letters, numbers, and underscore')
>>> IS_ALPHANUMERIC('this is not alphanumeric')('test!')
('test!', 'this is not alphanumeric')
>>> IS_ALPHANUMERIC(error_message='this is not alphanumeric')('test!')
('test!', 'this is not alphanumeric')
```

Hint The DAL validators are well documented inside the python source code. You can easily check it by yourself for all the details!

```
from pydal import validators
dir(validators) # get the list of all validators
help(validators.IS_URL) # get specific help for the IS_URL validator
```

12.6.1 Text format validators

IS_ALPHANUMERIC

This validator checks that a field value contains only characters in the ranges a-z, A-Z, 0-9, and underscores.

```
requires = IS_ALPHANUMERIC(error_message='must be alphanumeric!')
```

IS_LOWER

This validator never returns an error. It just converts the value to lower case.

```
requires = IS_LOWER()
```

IS_UPPER

This validator never returns an error. It converts the value to upper case.

```
requires = IS_UPPER()
```

IS_EMAIL

It checks that the field value looks like an email address. It does not try to send email to confirm.

```
requires = IS_EMAIL(error_message='invalid email!')
```

IS_MATCH

This validator matches the value against a regular expression and returns an error if it does not match. Here is an example of usage to validate a US zip code:

```
requires = IS_MATCH('^\d{5}(-\d{4})?$',
    error_message='not a zip code')
```

Here is an example of usage to validate an IPv4 address (note: the IS_IPV4 validator is more appropriate for this purpose):

```
requires = IS_MATCH('^\d{1,3}(\.\d{1,3}){3}$',
        error_message='not an IP address')
```

Here is an example of usage to validate a US phone number:

```
requires = IS_MATCH('^1?((-)\d{3}-?|\(\d{3}\))\d{3}-?\d{4}$',
        error_message='not a phone number')
```

For more information on Python regular expressions, refer to the official Python documentation.

`IS_MATCH` takes an optional argument `strict` which defaults to `False`. When set to `True` it only matches the whole string (from the beginning to the end):

```
>>> IS_MATCH('ab', strict=False)('abc')
('abc', None)
>>> IS_MATCH('ab', strict=True)('abc')
('abc', 'Invalid expression')
```

`IS_MATCH` takes an other optional argument `search` which defaults to `False`. When set to `True`, it uses regex method `search` instead of method `match` to validate the string.

`IS_MATCH('...', extract=True)` filters and extract only the first matching substring rather than the original value.

IS_LENGTH

Checks if length of field's value fits between given boundaries. Works for both text and file inputs.

Its arguments are:

- maxsize: the maximum allowed length / size (has default = 255)
- minsize: the minimum allowed length / size

Examples: Check if text string is shorter than 16 characters:

```
>>> IS_LENGTH(15)('example string')
('example string', None)
>>> IS_LENGTH(15)('example long string')
('example long string', 'Enter from 0 to 15 characters')
>>> IS_LENGTH(15)('33')
('33', None)
>>> IS_LENGTH(15)(33)
('33', None)
```

Check if uploaded file has size between 1KB and 1MB:

```
INPUT(_type='file', _name='name', requires=IS_LENGTH(1048576, 1024))
```

For all field types except for files, it checks the length of the value. In the case of files, the value is a `cgi.FieldStorage`, so it validates the length of the data in the file, which is the behavior one might intuitively expect.

IS_URL

Rejects a URL string if any of the following is true:

- The string is empty or None
- The string uses characters that are not allowed in a URL
- The string breaks any of the HTTP syntactic rules
- The URL scheme specified (if one is specified) is not 'http' or 'https'
- The top-level domain (if a host name is specified) does not exist

(These rules are based on `RFC 2616`)

This function only checks the URL's syntax. It does not check that the URL points to a real document, for example, or that it otherwise makes semantic sense. This function does automatically prepend 'http://' in front of a URL in the case of an abbreviated URL (e.g. 'google.ca'). If the parameter `mode='generic'` is used, then this function's behavior changes. It then rejects a URL string if any of the following is true:

- The string is empty or None
- The string uses characters that are not allowed in a URL

- The URL scheme specified (if one is specified) is not valid

(These rules are based on `RFC 2396`)

The list of allowed schemes is customizable with the allowed_schemes parameter. If you exclude None from the list, then abbreviated URLs (lacking a scheme such as 'http') will be rejected.

The default prepended scheme is customizable with the prepend_scheme parameter. If you set prepend_scheme to None, then prepending will be disabled. URLs that require prepending to parse will still be accepted, but the return value will not be modified.

IS_URL is compatible with the Internationalized Domain Name (IDN) standard specified in `RFC 3490`). As a result, URLs can be regular strings or unicode strings. If the URL's domain component (e.g. google.ca) contains non-US-ASCII letters, then the domain will be converted into Punycode (defined in `RFC 3492`). IS_URL goes a bit beyond the standards, and allows non-US-ASCII characters to be present in the path and query components of the URL as well. These non-US-ASCII characters will be encoded. For example, space will be encoded as'%20'. The unicode character with hex code 0x4e86 will become '%4e%86'.

Examples:

```
requires = IS_URL())
requires = IS_URL(mode='generic')
requires = IS_URL(allowed_schemes=['https'])
requires = IS_URL(prepend_scheme='https')
requires = IS_URL(mode='generic',
             allowed_schemes=['ftps', 'https'],
             prepend_scheme='https')
```

IS_SAFE

```
requires = IS_SAFE(error_message='Unsafe Content')
requires = IS_SAFE(mode="sanitize")
requires = IS_SAFE(sanitizer=lambda text: str(XML(text, sanitize=True)))
```

This validators is for text fields that should contain HTML and may contain invalid tags (script, ember, object, iframe). It works by trying to sanitize the content and either provide an error (mode="error") or replacing the content with the sanitized one (mode="sanitize"). You can specify the error message, the mode, and provide your own sanitizer.

IS_SLUG

```
requires = IS_SLUG(maxlen=80, check=False, error_message='must be slug')
```

If `check` is set to `True` it check whether the validated value is a slug (allowing only alphanumeric characters and non-repeated dashes).

If `check` is set to `False` (default) it converts the input value to a slug.

IS_JSON

```
requires = IS_JSON(error_message='Invalid json', native_json=False)
```

This validator checks that a field value is in JSON format.

If `native_json` is set to `False` (default) it converts the input value to the serialized value otherwise the input value is left unchanged.

12.6.2 Date and time validators

IS_TIME

This validator checks that a field value contains a valid time in the specified format.

```
requires = IS_TIME(error_message='must be HH:MM:SS!')
```

IS_DATE

This validator checks that a field value contains a valid date in the specified format. It is good practice to specify the format using the translation operator, in order to support different formats in different locales.

```
requires = IS_DATE(format=T('%Y-%m-%d'),
    error_message='must be YYYY-MM-DD!')
```

For the full description on % directives look under the IS_DATETIME validator.

IS_DATETIME

This validator checks that a field value contains a valid datetime in the specified format. It is good practice to specify the format using the translation operator, in order to support different formats in different locales.

```
requires = IS_DATETIME(format=T('%Y-%m-%d %H:%M:%S'),
                error_message='must be YYYY-MM-DD HH:MM:SS!')
```

The following symbols can be used for the format string (this shows the symbol, their meaning, and an example string):

```
%Y  year with century (e.g. '1963')
%y  year without century (e.g. '63')
%d  day of the month (e.g. '28')
%m  month (e.g '08')
%b  abbreviated month name (e.g.'Aug')
%B  full month name (e.g. 'August')
%H  hour (24-hour clock, e.g. '14')
%I  hour (12-hour clock, e.g. '02')
%p  either 'AM' or 'PM'
%M  minute (e.g. '30')
%S  second (e.g. '59')
```

IS_DATE_IN_RANGE

Works very much like the previous validator but allows to specify a range:

```
requires = IS_DATE_IN_RANGE(format=T('%Y-%m-%d'),
                minimum=datetime.date(2008, 1, 1),
                maximum=datetime.date(2009, 12, 31),
                error_message='must be YYYY-MM-DD!')
```

For the full description on % directives look under the IS_DATETIME validator.

IS_DATETIME_IN_RANGE

Works very much like the previous validator but allows to specify a range:

```
requires = IS_DATETIME_IN_RANGE(format=T('%Y-%m-%d %H:%M:%S'),
                minimum=datetime.datetime(2008, 1, 1, 10, 30),
                maximum=datetime.datetime(2009, 12, 31, 11, 45),
                error_message='must be YYYY-MM-DD HH:MM::SS!')
```

For the full description on % directives look under the IS_DATETIME validator.

12.6.3 Range, set and equality validators

IS_EQUAL_TO

Checks whether the validated value is equal to a given value (which can be a variable):

```
requires = IS_EQUAL_TO(request.vars.password,
```

```
                            error_message='passwords do not match')
```

IS_NOT_EMPTY

This validator checks that the content of the field value is neither None nor an empty string nor an empty list. A string value is checked for after a `.strip()`.

```
requires = IS_NOT_EMPTY(error_message='cannot be empty!')
```

You can provide a regular expression for the matching of the empty string.

```
requires = IS_NOT_EMPTY(error_message='Enter a value', empty_regex='NULL(?i)')
```

IS_NULL_OR

Deprecated, an alias for `IS_EMPTY_OR` described below.

IS_EMPTY_OR

Sometimes you need to allow empty values on a field along with other requirements. For example a field may be a date but it can also be empty. The `IS_EMPTY_OR` validator allows this:

```
requires = IS_EMPTY_OR(IS_DATE())
```

An empty value is either None or an empty string or an empty list. A string value is checked for after a `.strip()`.

You can provide a regular expression for the matching of the empty string with the `empty_regex` argument (like for IS_NOT_EMPTY validator).

You may also specify a value to be used for the empty case.

```
requires = IS_EMPTY_OR(IS_ALPHANUMERIC(), null='anonymous')
```

IS_EXPR

This validator let you express a general condition by means of a callable which takes a value to validate and returns the error message or `None` to accept the input value.

```
requires = IS_EXPR(lambda v: T('not divisible by 3') if int(v) % 3 else None)
```

Notice that returned message will not be translated if you do not arrange otherwise.

For backward compatibility the condition may be expressed as a string containing a logical expression in terms of a variable value. It validates a field value if the expression evaluates to `True`.

```
requires = IS_EXPR('int(value) % 3 == 0',
            error_message='not divisible by 3')
```

One should first check that the value is an integer so that an exception will not occur.

```
requires = [IS_INT_IN_RANGE(0, None),
            IS_EXPR(lambda v: T('not divisible by 3') if v % 3 else None)]
```

IS_DECIMAL_IN_RANGE

```
INPUT(_type='text', _name='name', requires=IS_DECIMAL_IN_RANGE(0, 10, dot="."))
```

It converts the input into a Python Decimal or generates an error if the decimal does not fall within the specified inclusive range. The comparison is made with Python Decimal arithmetic.

The minimum and maximum limits can be None, meaning no lower or upper limit, respectively.

The `dot` argument is optional and allows you to internationalize the symbol used to separate the decimals.

IS_FLOAT_IN_RANGE

Checks that the field value is a floating point number within a definite range, `0 <= value <= 100` in the following example:

```
requires = IS_FLOAT_IN_RANGE(0, 100, dot=".",
                             error_message='negative or too large!')
```

The `dot` argument is optional and allows you to internationalize the symbol used to separate the decimals.

IS_INT_IN_RANGE

Checks that the field value is an integer number within a definite range,
> `0 <= value < 100` in the following example:

```
requires = IS_INT_IN_RANGE(0, 100,
                           error_message='negative or too large!')
```

IS_IN_SET

This validator will automatically set the form field to an option field (ie, with a drop-down menu).

`IS_IN_SET` checks that the field values are in a set:

```
requires = IS_IN_SET(['a', 'b', 'c'], zero=T('choose one'),
            error_message='must be a or b or c')
```

The zero argument is optional and it determines the text of the option selected by default, an option which is not accepted by the `IS_IN_SET` validator itself. If you do not want a "choose one" option, set `zero=None`.

The elements of the set can be combined with a numerical validator, as long as IS_IN_SET is first in the list. Doing so will force conversion by the last validator to the numerical type. So, IS_IN_SET can be followed by `IS_INT_IN_RANGE` (which converts the value to int) or `IS_FLOAT_IN_RANGE` (which converts the value to float). For example:

```
requires = [ IS_IN_SET([2, 3, 5, 7], error_message='must be prime and less than
10'),
            IS_INT_IN_RANGE(0, None) ]
```

Checkbox validation

To force a filled-in form checkbox (such as an acceptance of terms and conditions), use this:

```
requires=IS_IN_SET(['ON'])
```

Dictionaries and tuples with IS_IN_SET

You may also use a dictionary or a list of tuples to make the drop down list more descriptive:

```
# Dictionary example:
requires = IS_IN_SET({'A':'Apple', 'B':'Banana', 'C':'Cherry'}, zero=None)

# List of tuples example:
requires = IS_IN_SET([('A', 'Apple'), ('B', 'Banana'), ('C', 'Cherry')])
```

Sorted options

To keep the options alphabetically sorted by their labels into the drop down list, use the `sort` argument with IS_IN_SET.

```
IS_IN_SET([('H', 'Hulk'), ('S', 'Superman'), ('B', 'Batman')], sort=True)
```

The `IS_IN_SET` validator has an optional attribute `multiple=False`. If set to True, multiple values can be stored in one field. The field should be of type `list:integer` or `list:string` as discussed in Section 7.14.1. An explicit example of tagging is discussed there. We strongly suggest using the jQuery multiselect plugin to render multiple fields.

Note that when `multiple=True`, IS_IN_SET will accept zero or more values, i.e. it will accept the field when nothing has been selected. `multiple` can also be a tuple of the form `(a, b)` where `a` and `b` are the minimum and (exclusive) maximum number of items that can be selected respectively.

12.6.4 Complexity and security validators

IS_STRONG

Enforces complexity requirements on a field (usually a password field).

Example:

```
requires = IS_STRONG(min=10, special=2, upper=2)
```

where:

- `min` is minimum length of the value
- `special` is the minimum number of required special characters, by default special characters are any of the following `^!!@#$%^&*()_+-=?<>,.:;{}[]|` (you can customize these using `specials = '...'`)
- `upper` is the minimum number of upper case characters

other accepted arguments are:

- `invalid` for the list of forbidden characters, by default `invalid=' "'`
- `max` for the maximum length of the value
- `lower` for the minimum number of lower case characters
- `number` for the minimum number of digit characters

Obviously you can provide an `error_message` as for any other validator, although IS_STRONG is clever enough to provide a clear message to describe the validation failure.

A special argument you can use is `entropy`, that is a minimum value for the complexity of the value to accept (a number), experiment this with:

```
>>> IS_STRONG(entropy=100.0)('hello')
('hello', Entropy (24.53) less than required (100.0))
```

Notice that if the argument `entropy` is not given then IS_STRONG implicitly sets the following defaults: `min = 8`, `upper = 1`, `lower = 1`, `number = 1`, `special = 1` which otherwise are all sets to `None`.

CRYPT

This is also a filter. It performs a secure hash on the input and it is used to prevent passwords from being passed in the clear to the database.

```
requires = CRYPT()
```

By default, CRYPT uses 1000 iterations of the pbkdf2 algorithm combined with SHA512 to produce a 20-byte-long hash. Old versions of web2py used md5 or HMAC+SHA512 depending on whether a key was specified or not.

If a key is specified, CRYPT uses the HMAC algorithm. The key may contain a prefix that determines

the algorithm to use with HMAC, for example SHA512:

```
requires = CRYPT(key='sha512:thisisthekey')
```

This is the recommended syntax. The key must be a unique string associated with the database used. The key can never be changed. If you lose the key, the previously hashed values become useless. By default, CRYPT uses random salt, such that each result is different. To use a constant salt value, specify its value:

```
requires = CRYPT(salt='mysaltvalue')
```

Or, to use no salt:

```
requires = CRYPT(salt=False)
```

The CRYPT validator hashes its input, and this makes it somewhat special. If you need to validate a password field before it is hashed, you can use CRYPT in a list of validators, but must make sure it is the last of the list, so that it is called last. For example:

```
requires = [IS_STRONG(), CRYPT(key='sha512:thisisthekey')]
```

CRYPT also takes a `min_length` argument, which defaults to zero.

The resulting hash takes the form `alg$salt$hash`, where `alg` is the hash algorithm used, `salt` is the salt string (which can be empty), and `hash` is the algorithm's output. Consequently, the hash is self-identifying, allowing, for example, the algorithm to be changed without invalidating previous hashes. The key, however, must remain the same.

12.6.5 Special type validators

IS_LIST_OF

This validator helps you to ensure length limits on values of type list, for this purpose use its `minimum`, `maximum`, and `error_message` arguments, for example:

```
requires = IS_LIST_OF(minimum=2)
```

A list value may comes from a form containing multiple fields with the same name or a multiple selection box. Note that this validator automatically converts a non-list value into a single valued list:

```
>>> IS_LIST_OF()('hello')
(['hello'], None)
```

If the first argument of IS_LIST_OF is another validator, then it applies the other validator to each element of the list. A typical usage is validation of a `list:` type field, for example:

```
Field('emails', 'list:string', requires=IS_LIST_OF(IS_EMAIL()), ...)
```

IS_LIST_OF_EMAILS

This validator is specifically designed to work with the following field:

```
Field('emails', 'list:string',
    widget=SQLFORM.widgets.text.widget,
  requires=IS_LIST_OF_EMAILS(),
  represent=lambda v, r:
      XML(', '.join([A(x, _href='mailto:'+x).xml() for x in (v or [])]))
  )
```

Notice that due to the `widget` customization this field will be rendered by a textarea in SQLFORMs (see next [[Widgets #Widgets]] section). This let you insert and edit multiple emails in a single input field (very much like normal mail client programs do), separating each email address with `,`, `;`, and blanks (space, newline, and tab characters). As a consequence now we need a validator which is able to operate on a single value input and a way to split the validated value into a list to be next processed

by DAL. This is accomplished by the `requires` validator. As alternative to `filter_in`, you can pass the following function to the `onvalidation` argument of form `accepts`, `process`, or `validate` method:

```
def emails_onvalidation(form):
    form.vars.emails = IS_LIST_OF_EMAILS.split_emails.findall(form.vars.emails)
```

The effect of the `represent` argument (at lines 6 and 7) is to add a "mailto:…" link to each email address when the record is rendered in HTML pages.

ANY_OF

This validator takes a list of validators and accepts a value if any of the validators in the list does (i.e. it acts like a logical OR with respect to given validators).

```
requires = ANY_OF([IS_ALPHANUMERIC(), IS_EMAIL()])
```

When none of the validators accepts the value you get the error message form the last attempted one (the last in the list), you can customize the error message as usual:

```
>>> ANY_OF([IS_ALPHANUMERIC(), IS_EMAIL()])('@ab.co')
('@ab.co', 'Enter a valid email address')
>>> ANY_OF([IS_ALPHANUMERIC(), IS_EMAIL()],
...        error_message='Enter login or email')('@ab.co')
('@ab.co', 'Enter login or email')
```

IS_IMAGE

This validator checks if a file uploaded through the file input was saved in one of the selected image formats and has dimensions (width and height) within given limits.

It does not check for maximum file size (use IS_LENGTH for that). It returns a validation failure if no data was uploaded. It supports the file formats BMP, GIF, JPEG, PNG, and it does not require the Python Imaging Library.

Code parts taken from ref.``source1``:cite

It takes the following arguments: - extensions: iterable containing allowed image file extensions in lowercase - maxsize: iterable containing maximum width and height of the image - minsize: iterable containing minimum width and height of the image

Use (-1, -1) as minsize to bypass the image-size check.

Here are some Examples: - Check if uploaded file is in any of supported image formats:

```
requires = IS_IMAGE()
```

- Check if uploaded file is either JPEG or PNG:

```
requires = IS_IMAGE(extensions=('jpeg', 'png'))
```

- Check if uploaded file is PNG with maximum size of 200x200 pixels:

```
requires = IS_IMAGE(extensions=('png'), maxsize=(200, 200))
```

Note: on displaying an edit form for a table including `requires = IS_IMAGE()`, a `delete` checkbox will NOT appear because to delete the file would cause the validation to fail. To display the `delete` checkbox use this validation:

```
requires = IS_EMPTY_OR(IS_IMAGE())
```

IS_FILE

Checks if name and extension of file uploaded through file input matches given criteria.

Does *not* ensure the file type in any way. Returns validation failure if no data was uploaded.

Its arguments are:

- filename: string/compiled regex or a list of strings/regex of valid filenames
- extension: string/compiled regex or a list of strings/regex of valid extensions
- lastdot: which dot should be used as a filename / extension separator: `True` indicates last dot (e.g., "file.tar.gz" will be broken in "file.tar" + "gz") while `False` means first dot (e.g., "file.tar.gz" will be broken into "file" + "tar.gz").
- case: 0 means keep the case; 1 means transform the string into lowercase (default); 2 means transform the string into uppercase.

If there is no dot present, extension checks will be done against empty string and filename checks against whole value.

Examples: Check if file has a pdf extension (case insensitive):

```
INPUT(_type='file', _name='name',
        requires=IS_FILE(extension='pdf'))
```

Check if file is called 'thumbnail' and has a jpg or png extension (case insensitive):

```
INPUT(_type='file', _name='name',
        requires=IS_FILE(filename='thumbnail',
        extension=['jpg', 'png']))
```

Check if file has a tar.gz extension and name starting with backup:

```
INPUT(_type='file', _name='name',
        requires=IS_FILE(filename=re.compile('backup.*'),
        extension='tar.gz', lastdot=False))
```

Check if file has no extension and name matching README (case sensitive):

```
INPUT(_type='file', _name='name',
    requires=IS_FILE(filename='README',
    extension='', case=0)
```

IS_UPLOAD_FILENAME

This is the older implementation for checking files, included for backwards compatibility. For new applications, use `IS_FILE()`.

This validator checks if the name and extension of a file uploaded through the file input matches the given criteria.

It does not ensure the file type in any way. Returns validation failure if no data was uploaded.

Its arguments are:

- filename: filename (before dot) regex.
- extension: extension (after dot) regex.
- lastdot: which dot should be used as a filename / extension separator: `True` indicates last dot (e.g., "file.tar.gz" will be broken in "file.tar" + "gz") while `False` means first dot (e.g., "file.tar.gz" will be broken into "file" + "tar.gz").
- case: 0 means keep the case; 1 means transform the string into lowercase (default); 2 means transform the string into uppercase.

If there is no dot present, extension checks will be done against an empty string and filename checks will be done against the whole value.

Examples:

Check if file has a pdf extension (case insensitive):

```
requires = IS_UPLOAD_FILENAME(extension='pdf')
```

Check if file has a tar.gz extension and name starting with backup:

```
requires = IS_UPLOAD_FILENAME(filename='backup.*', extension='tar.gz',
lastdot=False)
```

Check if file has no extension and name matching README (case sensitive):

```
requires = IS_UPLOAD_FILENAME(filename='^README$', extension='^$', case=0)
```

IS_IPV4

This validator checks if a field's value is an IP version 4 address in decimal form. Can be set to force addresses from a certain range.

IPv4 regex taken from `regexlib`. The signature for the `IS_IPV4` constructor is the following:

```
IS_IPV4(minip='0.0.0.0', maxip='255.255.255.255', invert=False,
        is_localhost=None, is_private=None, is_automatic=None,
        error_message='Enter valid IPv4 address')
```

Where:

- `minip` is the lowest allowed address
- `maxip` is the highest allowed address
- `invert` is a flag to invert allowed address range, i.e. if set to True allows addresses only from outside of given range; note that range boundaries are not matched this way

You can pass an IP address either as a string (e.g. '192.168.0.1') or as a list or tuple of 4 integers (e.g. [192, 168, 0, 1]).

To check for multiple address ranges pass to `minip` and `maxip` a list or tuple of boundary addresses, for example to allow only addresses between '192.168.20.10' and '192.168.20.19' or between '192.168.30.100' and '192.168.30.199' use:

```
requires = IS_IPV4(minip=('192.168.20.10', '192.168.30.100'),
                   maxip=('192.168.20.19', '192.168.30.199'))
```

Notice that only a range for which both lower and upper limits are set is configured, that is the number of configured ranges is determined by the shorter of the iterables passed to `minip` and `maxip`.

The arguments `is_localhost`, `is_private`, and `is_automatic` accept the following values:

- `None` to ignore the option
- `True` to force the option
- `False` to forbid the option

The option meanings are:

- `is_localhost`: match localhost address (127.0.0.1)
- `is_private`: match address in 172.16.0.0 - 172.31.255.255 and 192.168.0.0 - 192.168.255.255 ranges
- `is_automatic`: match address in 169.254.0.0 - 169.254.255.255 range

Examples:

Check for valid IPv4 address:

```
requires = IS_IPV4()
```

Check for valid private network IPv4 address:

```
requires = IS_IPV4(minip='192.168.0.1', maxip='192.168.255.255')
```

IS_IPV6

This validator checks if a field's value is an IP version 6 address.

The signature for the `IS_IPV6` constructor is the following:

```
IS_IPV6(is_private=None,
        is_link_local=None,
        is_reserved=None,
        is_multicast=None,
        is_routeable=None,
        is_6to4=None,
        is_teredo=None,
        subnets=None,
        error_message='Enter valid IPv6 address')
```

The arguments `is_private`, `is_link_local`, `is_reserved`, `is_multicast`, `is_routeable`, `is_6to4`, and `is_teredo` accept the following values:

- `None` to ignore the option
- `True` to force the option
- `False` to forbid the option, this does not work for `is_routeable`

The option meanings are:

- `is_private`: match an address allocated for private networks
- `is_link_local`: match an address reserved for link-local (i.e. in fe80::/10 range), this is a private network too (also matched by `is_private` above)
- `is_reserved`: match an address otherwise IETF reserved
- `is_multicast`: match an address reserved for multicast use (i.e. in ff00::/8 range)
- `is_6to4`: match an address that appear to contain a 6to4 embedded address (i.e. in 2002::/16 range)
- `is_teredo`: match a teredo address (i.e. in 2001::/32 range)

Forcing `is_routeable` (setting to True) is a shortcut to forbid (setting to False) `is_private`, `is_reserved`, and `is_multicast` all.

Use the `subnets` argument to pass a subnet or list of subnets to check for address membership, this way an address must be a subnet member to validate.

Examples:

Check for valid IPv6 address:

```
requires = IS_IPV6()
```

Check for valid private network IPv6 address:

```
requires = IS_IPV6(is_link_local=True)
```

Check for valid IPv6 address in subnet:

```
requires = IS_IPV6(subnets='fb00::/8')
```

IS_IPADDRESS

This validator checks if a field's value is an IP address (either version 4 or version 6). Can be set to force addresses from within a specific range. Checks are done using the appropriate `IS_IPV4` or `IS_IPV6` validator.

The signature for the `IS_IPADDRESS` constructor is the following:

```
IS_IPADDRESS(minip='0.0.0.0', maxip='255.255.255.255', invert=False,
             is_localhost=None, is_private=None, is_automatic=None,
             is_ipv4=None,
             is_link_local=None, is_reserved=None, is_multicast=None,
             is_routeable=None, is_6to4=None, is_teredo=None,
             subnets=None, is_ipv6=None,
             error_message='Enter valid IP address')
```

With respect to `IS_IPV4` and `IS_IPV6` validators the only added arguments are:

- `is_ipv4`, set to True to force version 4 or set to False to forbid version 4
- `is_ipv6`, set to True to force version 6 or set to False to forbid version 6

Refer to IS_IPV4 and IS_IPV6 validators for the meaning of other arguments.

Examples:

Check for valid IP address (both IPv4 and IPv6):

```
requires = IS_IPADDRESS()
```

Check for valid IP address (IPv6 only):

```
requires = IS_IPADDRESS(is_ipv6=True)
```

12.6.6 Other validators

CLEANUP

This is a filter. It never fails. By default it just removes all characters whose decimal ASCII codes are not in the list [10, 13, 32-127]. It always perform an initial strip (i.e. heading and trailing blank characters removal) on the value.

```
requires = CLEANUP()
```

You can pass a regular expression to decide what has to be removed, for example to clear all non-digit characters use:

```
>>> CLEANUP('[^\\d]')('Hello 123 world 456')
('123456', None)
```

12.6.7 Database validators

IS_NOT_IN_DB

Synopsis: `IS_NOT_IN_DB(db|set, 'table.field')`

Consider the following example:

```
db.define_table('person', Field('name'))
db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

It requires that when you insert a new person, his/her name is not already in the database, `db`, in the field `person.name`.

A set can be used instead of `db`.

As with all other validators this requirement is enforced at the form processing level, not at the database level. This means that there is a small probability that, if two visitors try to concurrently insert records with the same person.name, this results in a race condition and both records are accepted. It is therefore safer to also inform the database that this field should have a unique value:

```
db.define_table('person', Field('name', unique=True))
db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

Now if a race condition occurs, the database raises an OperationalError and one of the two inserts is rejected.

The first argument of `IS_NOT_IN_DB` can be a database connection or a Set. In the latter case, you would be checking only the set defined by the Set.

A complete argument list for `IS_NOT_IN_DB()` is as follows:

```
IS_NOT_IN_DB(dbset, field, error_message='value already in database or empty',
             allowed_override=[], ignore_common_filters=True)
```

The following code, for example, does not allow registration of two persons with the same name within 10 days of each other:

```
import datetime
now = datetime.datetime.today()
db.define_table('person',
    Field('name'),
    Field('registration_stamp', 'datetime', default=now))
recent = db(db.person.registration_stamp > now-datetime.timedelta(10))
db.person.name.requires = IS_NOT_IN_DB(recent, 'person.name')
```

IS_IN_DB

Synopsis: `IS_IN_DB(db|set, 'table.value_field', '%(representing_field)s', zero='choose one')` where the third and fourth arguments are optional.

`multiple=` is also possible if the field type is a list. The default is False. It can be set to True or to a tuple (min, max) to restrict the number of values selected. So `multiple=(1, 10)` enforces at least one and at most ten selections.

Other optional arguments are discussed below.

Example Consider the following tables and requirement:

```
db.define_table('person', Field('name', unique=True))
db.define_table('dog', Field('name'), Field('owner', db.person))
db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s',
                                 zero=T('choose one'))
```

the IS_IN_DB requirement could also be written to use a Set instead of db

```
db.dog.owner.requires = IS_IN_DB(db(db.person.id > 10), 'person.id', '%(name)s',
                                 zero=T('choose one'))
```

It is enforced at the level of dog INSERT/UPDATE/DELETE forms. This example requires that a `dog.owner` be a valid id in the field `person.id` in the database `db`. Because of this validator, the `dog.owner` field is represented as a drop-down list. The third argument of the validator is a string that describes the elements in the drop-down list, this is passed to the `label` argument of the validator. In the example you want to see the person `%(name)s` instead of the person `%(id)s`. `%(...)s` is replaced by the value of the field in brackets for each record. Other accepted values for the `label` are a Field instance (e.g. you could use db.person.name instead of '%(name)s') or even a callable that takes a row and returns the description for the option.

The `zero` option works very much like for the `IS_IN_SET` validator.

Other optional arguments accepted by `IS_IN_DB` are: `orderby`, `groupby`, `distinct`, `cache`, and `left`; these are passed to the db select (see Section 7.9.6 on the DAL chapter).

Notice that `groupby`, `distinct`, and `left` do not apply to Google App Engine.

To alphabetically sort the options listed in the drop-down list you can set the `sort` argument to `True` (sorting is case-insensitive), this may be usefull when no `orderby` is feasible or practical.

The first argument of the validator can be a database connection or a DAL Set, as in `IS_NOT_IN_DB`. This can be useful for example when wishing to limit the records in the drop-down list. In this example, we use `IS_IN_DB` in a controller to limit the records dynamically each time the controller is called:

```python
def index():
    (...)
    query = (db.table.field == 'xyz') # in practice 'xyz' would be a variable
    db.table.field.requires = IS_IN_DB(db(query), ...)
    form = Form(...)
    if form.process().accepted: ...
    (...)
```

If you want the field validated, but you do not want a drop-down, you must put the validator in a list.

```python
db.dog.owner.requires = [IS_IN_DB(db, 'person.id', '%(name)s')]
```

Occasionally you want the drop-down (so you do not want to use the list syntax above) yet you want to use additional validators. For this purpose the `IS_IN_DB` validator takes an extra argument `_and` that can point to a list of other validators applied if the validated value passes the `IS_IN_DB` validation. For example to validate all dog owners in db that are not in a subset:

```python
subset = db(db.person.id > 100)
db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s',
                                 _and=IS_NOT_IN_DB(subset, 'person.id'))
```

`IS_IN_DB` and Tagging

The `IS_IN_DB` validator has an optional attribute `multiple=False`. If set to `True` multiple values can be stored in one field. This field should be of type `list:reference` as discussed in Section 7.14.1. An explicit example of tagging is discussed there. Multiple references are handled automatically in create and update forms, but they are transparent to the DAL. We strongly suggest using the jQuery multiselect plugin to render multiple fields.

12.6.8 Validation functions

In order to explicitly define a validation function, we pass to the `validation` parameter 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:

```python
from py4web import Field
from py4web.utils.form import Form, FormStyleBulma
from pydal.validators import IS_INT_IN_RANGE

def custom_check(form):
    if not 'name' in form.errors and len(form.vars['name']) < 4
        form.errors['name'] = T("too short")

@action('form_example', method=['GET', 'POST'])
@action.uses('form_example.html', session)
def form_example():
    form = Form(db.thing, validation=custom_check)
    if form.accepted:
        redirect(URL('index'))
    return dict(form=form)
```

Authentication and authorization

Strong authentication and authorization methods are vital for a modern, multiuser web application. While they are often used interchangeably, authentication and authorization are separate processes:

- Authentication confirms that users are who they say they are
- Authorization gives those users permission to access a resource

13.1 Authentication using Auth

py4web comes with a an object `Auth` and a system of plugins for user authentication. It has the same name as the corresponding web2py one and serves the same purpose but the API and internal design is very different.

The _scaffold application provides a guideline for its standard usage. By default it uses a local SQLite database and allows creating new users, login and logout. Notice that if you don't configure it, you have to manually approve new users (by visiting the link logged on the console or by directly editing the database).

To use the Auth object, 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` which is created if missing. The `auth_user` table has 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).

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.

13.1.1 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/templates/layout.html

The key section is in `layout.html` where (using the no.css framework) the menu actions are defined:

```
<ul>
   [[if globals().get('user'):]]
   <li>
   <a class="navbar-link is-primary">
      [[=globals().get('user',{}).get('email')]]
   </a>
   <ul>
      <li><a href="[[=URL('auth/profile')]]">Edit Profile</a></li>
      [[if 'change_password' in
globals().get('actions',{}).get('allowed_actions',{}):]]
         <li><a href="[[=URL('auth/change_password')]]">Change Password</a></li>
      [[pass]]
      <li><a href="[[=URL('auth/logout')]]">Logout</a></li>
   </ul>
   </li>
   [[else:]]
   <li>
   Login
   <ul>
      <li><a href="[[=URL('auth/register')]]">Sign up</a></li>
      <li><a href="[[=URL('auth/login')]]">Log in</a></li>
   </ul>
   </li>
   [[pass]]
</ul>
```

The menu is dynamic: on line 2 there is a check if the user is already defined (i.e. if the user has already logged on). In this case the email is shown in the top menu, plus the menu options `Edit Profile`, `Change Password` (optional) and `Logout`. Instead, if the user is not already logged on, from line 15 there are only the corresponding menu options allowed: `Sign up` and `Log in`.

Every menu option then redirects the user to the corresponding standard URL, which in turn activates the Auth action.

13.1.2 Using Auth inside actions

There two ways to use the Auth object in an action.

The first one does not force a login. With `@action.uses(auth)` we tell py4web that this action should have information about the user, trying to parse the session for a user session.

```python
@action('index')
@action.uses(auth)
def index():
    user = auth.get_user()
    return 'hello {first_name}'.format(**user) if user else 'not logged in'
```

The second one forces the login if needed:

```python
@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.

13.1.3 Two Factor Authentication

Two factor authentication (or Two-step verification) is a way of improving authentication security. When activated an extra step is added in the login process. In the first step, users are shown the standard username/password form. If they successfully pass this challenge by submitting the correct username and password, and two factor authentication is enabled for the user, the server will present a second form before logging them in.

There are a few Auth settings available to control how two factor authentication works.

The following can be specified on Auth instantiation:

- `two_factor_required`
- `two_factor_send`
- `two_factor_validate`

two_factor_required

When you pass a method name to the `two_factor_required` parameter you are telling py4web to call that method to determine whether or not this login should be use or bypass two factor authentication. If your method returns True, then this login requires two factor. If it returns False, two factor authentication is bypassed for this login.

Sample `two_factor_required` method

This example shows how to allow users that are on a specific network.

```python
def user_outside_network(user, request):
    import ipaddress

    networks = ["10.10.0.0/22"]

    ip_list = []
    for range in networks:
        ip_list.extend(ipaddress.IPv4Network(range))

    if ipaddress.IPv4Address(request.remote_addr) in ip_list:
        #  if the client address is in the network address list, then do NOT
        require MFA
        return False

    return True
```

two_factor_send

When two factor authentication is active, py4web can generate a 6 digit code (using random.randint) and makes it possible to send it to the user. How this code is sent, is up to you. The `two_factor_send` argument to the Auth class allows you to specify the method that sends the two factor code to the user.

This example shows how to send an email with the two factor code:

```python
def send_two_factor_email(user, code):
    try:
        auth.sender.send(
            to=[user.email],
            subject=f"Two factor login verification code",
            body=f"You're verification code is {code}",
            sender="from_address@youremail.com",
        )
    except Exception as e:
        print(e)
    return code
```

Notice that this method takes two arguments: the current user, and the code to be sent. Also notice this method can override the code and return a new one.

```
auth.param.two_factor_required = user_outside_network
auth.param.two_factor_send = send_two_factor_email
```

two_factor_validate

By default, py4web will validate the user input in the two factor form by comparing the code entered by the user with the code generated and sent using `two_factor_send`. However, sometimes it may be useful to define a custom validation of this user-entered code. For instance, if one would like to use the TOTP (or the Time-Based One-Time-Passwords) as the two factor authentication method, the validation requires comparing the code entered by the user with the value generated at the same time at the server side. Hence, it is not sufficient to generate that value earlier when showing the form (using for instance `two_factor_send` method), because by the time the user submits the form, the current valid value may already be different. Instead, this value should be generated when validating the form submitted by the user.

To accomplish such custom validation, the `two_factor_validate` method is available. It takes two arguments:

- the current user
- the code that was entered by the user into the two factor authentication form

The primary use-case for this method is validation of time-based passwords.

This example shows how to validate a time-based two factor code:

```python
def validate_code(user, code):
    try:
        # get the correct code from an external function
        correct_code = generate_time_based_code(user_id)
    except Exception as e:
        # return None to indicate that validation could not be performed
        return None

    # compare the value entered in the auth form with the correct code
    if code == correct_code:
        return True
    else:
        return False
```

The `validate_code` method must return one of three values:

- `True` - if the validation succeeded,
- `False` - if the validation failed,
- `None` - if the validation was not possible for any reason

Notice that - if defined - this method is _always_ called to validate the two factor authentication form. It is up to you to decide what kind of validation it does. If the returned value is `True`, the user input will be accepted as valid. If the returned value is `False` then the user input will be rejected as invalid, number of tries will be decreased by one, and user will be asked to try again. If the returned value is `None` the user input will be checked against the code generated with the use of `two_factor_send` method and the final result will depend on that comparison. In this case authentication will fail if `two_factor_send` method was not defined, and hence no code was sent to the user.

```
auth.param.two_factor_validate = validate_code
```

two_factor_tries

By default, the user has 3 attempts to pass two factor authentication. You can override this after using:

```
auth.param.two_factor_tries = 5
```

Once this is all setup, the flow for two factor authentication is:

- present the login page
-
  upon successful login and user passes two_factor_required

    - redirect to py4web auth/two_factor endpoint
    -
      if `two_factor_send` method has been defined:

        - generate 6 digit verification code
        - call `two_factor_send` to send the verification code to the user

    - display verification page where user can enter their code
    - if `two_factor_validate` method has been defined - call it to validate the user-entered code
    - upon successful verification, take user to _next_url that was passed to the login page

Important! If you filtered `ALLOWED_ACTIONS` in your app, make sure to whitelist the "two_factor" action so not to block the two factor API.

13.1.4 Auth Plugins

Plugins are defined in "py4web/utils/auth_plugins" and they have a hierarchical 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.

In the _scaffold/settings.py and _scaffold/common.py files you can see the default settings for the supported plugins.

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. 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.

Note by design PAM authentication using local users works fine only if py4web is run by root. Otherwise you can only authenticate the specific user that runs the py4web process.

LDAP

This is a common authentication method, especially using Microsoft Active Directory in enterprises.

```
from py4web.utils.auth_plugins.ldap_plugin import LDAPPlugin
LDAP_SETTING = {
    'mode': 'ad',
    'server': 'my.domain.controller',
    'base_dn': 'cn=Users,dc=domain,dc=com'
}
auth.register_plugin(LDAPPlugin(**LDAP_SETTINGS))
```

> **Warning** it needs the python-ldap module. On Ubuntu, you should also install some developer's libraries in advance with `sudo apt-get install libldap2-dev libsasl2-dev`.

OAuth2 with Google

```
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.

By default, Google OAuth stores the user's first name, last name, and email in the auth_user table—but not the profile picture. You can include the profile picture URL with just few lines of code added to common.py.

```
from py4web.utils.auth_plugins.oauth2google import OAuth2Google  # TESTED
...
auth = Auth(session, db, define_tables=False)
auth.extra_auth_user_fields = [
    Field('profile_picture', 'text', readable=False, writable=False)
]
...
OAuth2Google.maps['profile_picture'] = 'picture'
```

Once the profile picture URL is stored in auth_user, you can easily use it along with other user information.

OAuth2 with Facebook

```
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.

OAuth2 with Discord

```
from py4web.utils.auth_plugins.oauth2discord import OAuth2Discord
auth.register_plugin(OAuth2Discord(
    client_id=DISCORD_CLIENT_ID,
    client_secret=DISCORD_CLIENT_SECRET,
    callback_url="auth/plugin/oauth2discord/callback"))
```

To obtain a Discord client ID and secret, create an application at https://discord.com/developers/applications. You will also have to register your OAuth2 redirect URI in your created application, in the form of `http(s)://<your host>/<your app name>/auth/plugin/oauth2discord/callback`

Note As Discord users have no concept of first/last name, the user in the auth table will contain the Discord username as the first name and discriminator as the last name.

13.1.5 Auth API Plugins

There are two types of web APIs, those called by the browser for example by a single page web app, and those designed to be called by a different kind of program. Both of them may need tosupport authentication. The distintion is important because, in the case of the browser, there is no need to manage any authentication token as the browser already provides cookies and py4web uses cookies to handle seesions. If the user operating the browser is logged-in, when an API is called, the corresponding action already knows who the user is. No additional logic is necessary. In this case there there is no need for any kind of additional API token which would only diminuish the security provided by the cookie based session token.

When the API is to be accessed by a different program (for example a script) the story is different. There is no session and we do not want to ask the user for the password every time. The standard way to authenticate in this case is by issuing the user an API token, aka a string, which, when presented along with API request allows py4web to recognize the identity of the caller. This is also referred to as "Authentication bearer".

Py4web provides a plugin system that gives you a lot of flexibility but it also provides two practical plugins that are sufficient in most cases. The two plugins are called: SimpleTokenPlugin and JwtTokenPlugin. The first one of the two is recommended in most of the cases.

What all plugins have in common:

- They have a way for a user to create a token which is a string.
- When an HTTP(S) request is made to an action that @actiion.uses(auth) or @action.uses(auth.user) py4web will identify the user if the token is present, as if the user was logged-in.

What SimpleTokenPlugin and JwtTokenPlugin have in common:

- When an HTTP(S) request is made, the token must be put in the "Authentication" header. You will need to create your own plugin if you want to pass it in some other manner.
- Each user can create as many tokens as desired.
- Users can create tokens for other users if the application logic requires/allows it.

Unique features of SimpleTokenPlugin:

- A token is a UUID.
- Tokens can be managed serverside (created, deleted, expired, change expiration).
- Current tokens are stored in a adatabase table.

- The default table associates token with the owner and a textual description. Users can nevertheless provide their own table and add any desired metadata to tokens which the app can retrieve to distinguish different tokens from the same user. This is done by adding fields to the table.

- Under the hood veryfing a token requires a database query.

Unique features of JwtTokenPlugin:

- The token is an encrypted and digitally signed dict that stores the user_id and expiration.

- The author of the token can add any metadata to into the token at creation.

- The token is not stored anywhere serverside and there is no database table.

- Tokens can be created (and there is a function to do so) but they cannot be managed. The server cannot expire tokens or change expiration. This would require the tokens to validated against a database and that is exactely when the JwtTokenPlugin tries to avoid.

- The only way to expire a token is by changing the serverside secret using for validation so when a token is expired, all tokens are expired.

SimpleTokenPlugin are the recommended kind of tokens for most applications. JwtTokenPlugin are valuable when the expiration is short and known in advance and when avoiding a database lookup is very important, such as for actions that are very fast and one is willing to sacrifice a bit of security (serverside token expiration capability) in order to avoid database access.

Example of SimpleTokenPlugin

In common.py:

```
from py4web.utils.auth import SimpleTokenPlugin
simple_token_plugin = SimpleTokenPlugin(auth)
auth.token_plugins.append(simple_token_plugin)
```

You can optionally a `table=db.mytable` to a custom table. Otherwise it will create and use one called "auth_simple_token".

In controllers.py

```
@action("test_api")
@action.uses(auth.user)
def test_api():
    return {"hello": "world"}
```

Users can access this action if via a browser if they are logged in, without the token, of via API by providing a token.

```
curl http://127.0.0.1:8000/test1/test_api -H "Authorization: Bearer {token}"
```

In order to create and manage tokens you can use a grid. In controllers.py

```
@action("tokens")
def _():
    db.auth_simple_token.user_id.default = auth.user_id
    grid = Grid(db.auth_simple_token.user_id==auth.user_id, create=True,
deletable=True)
    return dict(grid=grid)
```

Example of JwtTokenPlugin

In common.py:

```
from py4web.utils.auth import SimpleTokenPlugin
jwt_token_plugin = JwtTokenPlugin(auth)
auth.token_plugins.append(jwt_token_plugin)
```

In controllers.py it works the same as SimpleTokenPlugin:

```
@action("test_api")
@action.uses(auth.user)
def test_api():
    return {"hello": "world"}
```

The token is also passed using the same header as in the previous example:

```
curl http://127.0.0.1:8000/test1/test_api -H "Authorization: Bearer {token}"
```

While you cannot manage tokens you still need a way to create them. You can create an action for example that, when called, gives you a new token. In controllers.py

```
@action("make_token")
@action.uses("generic.html", auth.user)
def make_token():
    return dict(token=jwt_token_plugin.make(
        auth.current_user,
        expiration=utcnow()+datetime.timedelta(days=10)))
```

Example of custom Token Plugin

A token plugin is just a class that, given a request, returns an associated user. For example here is a dumb and UNSAFE plugin that authorizes everybody as user 1 as long as the "Authentication" header is provided.

from py4web import request

```
class MyCustomTokenPlugin:
    def get_user(self):
        authorization = request.headers.get("Authentication")
        if authorization:
            return db.auth_user(1)
        return None

auth.token_plugins.append(MyCustomTokenPlugin())
```

13.2 Authorization using Tags

As already mentioned, authorization is the process of verifying what specific applications, files, and data a user has access to. This is accomplished in py4web using `Tags`, that we've already discovered on Section 7.7.7 in the DAL chapter.

13.2.1 Tags and Permissions

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. Developers are free to create their own logic on top of the tagging system.

Note 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.

To use the tagging system you first need to import the Tags module from `pydal.tools`. Then create a Tags object to tag a table:

```
from pydal.tools.tags import Tags
```

```
groups = Tags(db.auth_user, 'groups')
```

The tail_name parameter is optional and if not specified the 'default' value will be used. If you look at the database level, a new table will be created with a name equals to `tagged_db + '_tag_' + tail_name`, in this case `auth_user_tag_groups`:

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')
```

On the `auth_user_tagged_groups` this will produce two records with different groups assigned to the same user.id (the "Record ID" field):

Slashes at the beginning or the end of a tag are optional. All other chars are allowed on equal footing.

A common use case is **group based access control**. Here 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'. Else the user is in the correct group and 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've already seen a simple `requires_membership` fixture on :ref:`The Condition fixture`. It enables the following syntax:

```
groups = Tags(db.auth_user)
```

```
class requires_membership(Fixture):
    def __init__(self, group):
        self.__prerequisites__ = [auth.user] # you must have a user before you can
check
        self.group  = group # store the group when action defined
    def on_request(self, context): # will be called if the action is called
        if self.group not in groups.get(auth.user_id):
            raise HTTP(401) # check and do something


@action('index')
@action.uses(requires_membership('teacher'))
def index():
    return 'hello teacher'
```

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 searches 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.

13.2.2 Multiple Tags objects

Note One table can have multiple associated `Tags` objects. The name 'groups' here is completely arbitrary but has a specific semantic meaning. Different `Tags` objects are independent to each other. The limit to their use is your creativity.

For example you could create a table `auth_group`:

```
db.define_table('auth_group', Field('name'), Field('description'))
```

and two Tags attached to it:

```
groups = Tags(db.auth_user)
permissions = Tags(db.auth_groups)
```

Then create a 'zapper' record in `auth_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.

13.2.3 User Impersonation

Auth provides API that allow you to impersonate another user. Here is an example of an action to start impersonating and stop impersonating another user.

```python
@action("impersonate/{user_id:int}", method="GET")
@action.uses(auth.user)
def start_impersonating(user_id):
    if (not auth.is_impersonating() and
        user_id and
        user_id != auth.user_id and
        db(db.auth_user.id==user_id).count()):
        auth.start_impersonating(user_id, URL("index"))
    raise HTTP(404)

 @action("stop_impersonating", method="GET")
 @action.uses(auth)
 def stop_impersonating():
    if auth and auth.is_impersonating():
        auth.stop_impersonating(URL("index"))
    redirect(URL("index"))
```

Grid

py4web comes with a Grid object providing grid and CRUD (create, update and delete) capabilities. This allows you to quickly and safely provide an interface to your data. Since it's also highly customizable, it's the corner stone of most py4web's applications.

14.1 Key features

- Full CRUD with Delete Confirmation
- Click column heads for sorting - click again for DESC
- Pagination control
- Built in Search (can use search_queries OR search_form)
- Action Buttons - with or without text
- Pre and Post Action (add your own buttons to each row)
- Grid dates in local format
- Default formatting by type plus user overrides

Hint There is an excellent grid tutorial made by Jim Steil on https://github.com/jpsteil/grid_tutorial. You're strongly advised to check it for any doubt and for finding many precious examples, hints & tips.

14.2 Basic grid example

In this simple example we will make a grid over the superhero table.

Create a new minimal app called `grid`. Change it with the following content.

```python
# in grid/__init__.py
import os
from py4web import action, Field, DAL
from py4web.utils.grid import *
from py4web.utils.form import *
from yatl.helpers import A


# database definition
DB_FOLDER = os.path.join(os.path.dirname(__file__), 'databases')
if not os.path.isdir(DB_FOLDER):
    os.mkdir(DB_FOLDER)
```

```
db = DAL('sqlite://storage.sqlite', folder=DB_FOLDER)
db.define_table(
    'person',
    Field('superhero'),
    Field('name'),
    Field('job'))

# add example entries in db
if not db(db.person).count():
    db.person.insert(superhero='Superman', name='Clark Kent', job='Journalist')
    db.person.insert(superhero='Spiderman', name='Peter Park', job='Photographer')
    db.person.insert(superhero='Batman', name='Bruce Wayne', job='CEO')
    db.commit()

@action('index', method=['POST', 'GET'])
@action.uses('grid.html', db)
def index():
    grid = Grid(
            formstyle=FormStyleDefault, # FormStyleDefault, FormStyleBulma,
FormStyleBootstrap4, or FormStyleBootstrap5
            grid_class_style=GridClassStyle, # GridClassStyle or
GridClassStyleBulma or GridClassStyleBootstrap5
            icon_style=IconStyleFontawesome, # IconStyle, IconStyleFontawesome, or
IconStyleBootstrapIcons
            query=(db.person.id > 0),
            orderby=[db.person.name],
            search_queries=[['Search by Name', lambda val:
db.person.name.contains(val)]])

    return dict(grid=grid)
```

Add a new file templates/grid.html with this basic content:

```
[[=grid.render()]]
```

Then restart py4web. If you browse to http://127.0.0.1:8000/grid/index you'll get this result:

Its layout is quite minimal, but it's perfectly usable.

The main problem is that by default the **no.css** stylesheet is used, see here. But we've not loaded it!

Change the file templates/grid.html with this content:

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.14.0/css/all.min.css"
/>
    </head>
    <body>
        [[=grid.render()]]
    <body>
</html>
```

Then refresh the page.
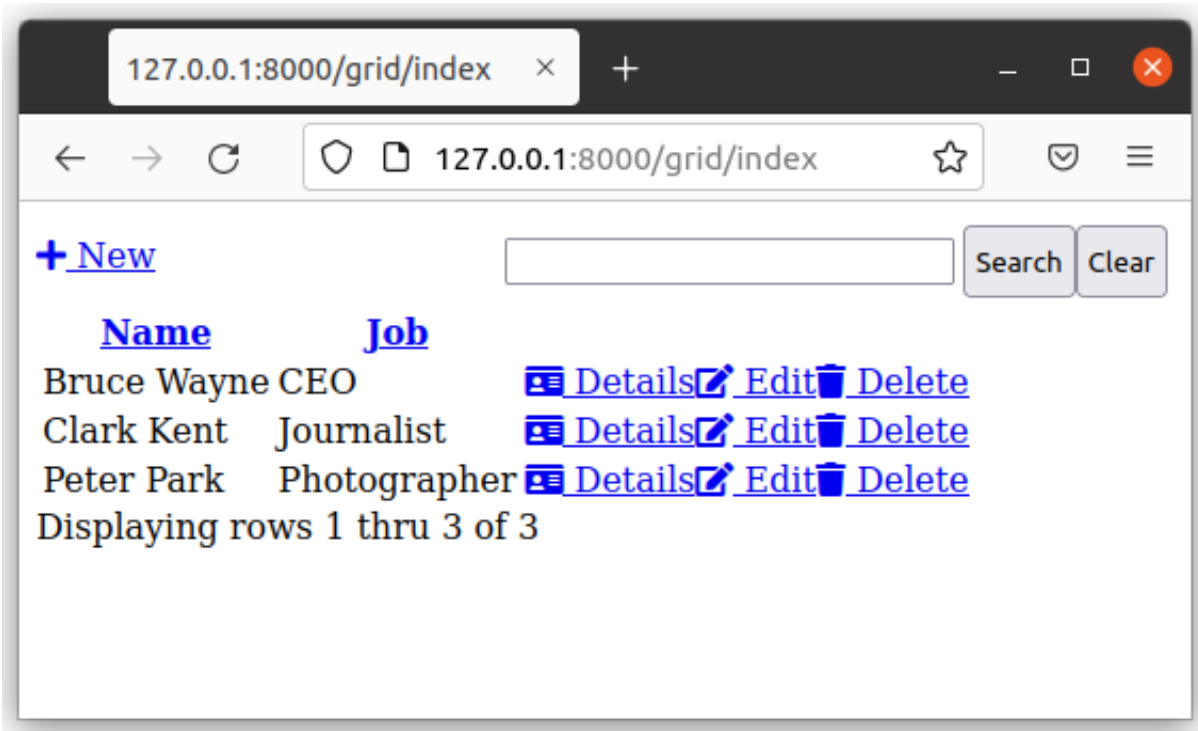
This is better now, with proper icons for Details, Edit and Delete actions.

We can also think about using the **bulma.css**, see here. In this case you need to change the grid object on __init__.py to:

```
formstyle=FormStyleBulma, # FormStyleDefault or
FormStyleBulma,FormStyleBootstrap4, or FormStyleBootstrap5
grid_class_style=GridClassStyleBulma, # GridClassStyle or GridClassStyleBulma or
GridClassStyleBootstrap5
```

Notice that in this case you need to import the corresponding python modules in advance (we've already done it on line 4 and 5 above). If you use the default no.css style instead, you don't need to manually import its style modules (and you even don't need the formstyle and grid_class_style parameters).

You also have to change the file templates/grid.html with this content:

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.9.3/css/bulma.min.css">
```

```
    </head>
    <body>
            [[=grid.render()]]
    <body>
</html>
```

Then refresh the page.
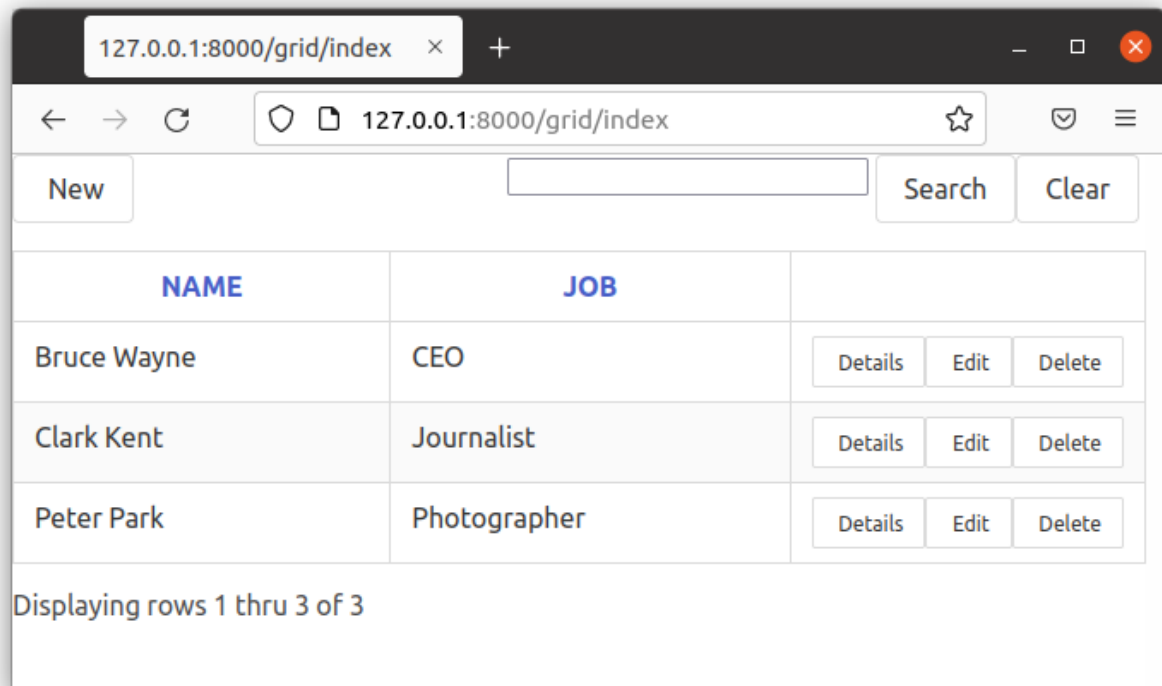
This is much better, isn't it?

Bootstrap4 and Bootstrap5 also have styles available, and you can change the icon style between Fontawesome, Bootstrap Icons, and a basic IconStyle you can write your own CSS for. More info in the section: Section 14.6

Note These are just minimal examples for showing how `grid` works internally. Normally you should start from a copy of the standard `_scaffold` app, with all the Session and Authentication stuff already defined. Also, you should follow the standard rules for code, like placing the db definition inside models.py and so on. Using standards will make your code simpler, safer and more maintainable.

Also, do not use grid objects directly on the root action of an app, because it does not add the 'index' route. So, in this example if you browse to http://127.0.0.1:8000/grid the main page is displayed fine but any contained action will lead to a non existent page.

In the Chapter 16 chapter you can find more examples, including a master/detail grid example written with **htmx**. And don't forget Jim Steil's detailed tutorial on https://github.com/jpsteil/grid_tutorial.

14.3 The Grid object

```
class Grid:
    def __init__(
        self,
        path,
```

```
    query,
    search_form=None,
    search_queries=None,
    columns=None,
    field_id=None,
    show_id=False,
    orderby=None,
    left=None,
    headings=None,
    create=True,
    details=True,
    editable=True,
    deletable=True,
    validation=None,
    pre_action_buttons=None,
    post_action_buttons=None,
    auto_process=True,
    rows_per_page=15,
    include_action_button_text=True,
    search_button_text="Filter",
    formstyle=FormStyleDefault,
    grid_class_style=GridClassStyle,
    icon_style=IconStyleFontawsome,
    T=lambda text: text,
):
```

- path: the route of this request

- query: pydal query to be processed

- search_form: py4web FORM to be included as the search form. If search_form is passed in then the developer is responsible for applying the filter to the query passed in. This differs from search_queries

- search_queries: list of query lists to use to build the search form. Ignored if search_form is used

- columns: list of fields or columns to display on the list page, see the Section 14.4 paragraph later. If blank, the table will use all readable fields of the searched 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 True to automatically generate the URL - set to False to not display the button

- details: URL to redirect to for displaying records - set to True to automatically generate the URL - set to False to not display the button (*)

- editable: URL to redirect to for editing records - set to True to automatically generate the URL - set to False to not display the button (*)

- deletable: URL to redirect to for deleting records - set to True to automatically generate the URL - set to False to not display the button (*)

- validation: optional validation function to pass to create and edit forms

- 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

- auto_process: Boolean - whether or not the grid should be processed immediately. If False, developer must call grid.process() once all params are setup

- rows_per_page: number of rows to display per page. Default 15

- include_action_button_text: boolean telling the grid whether or not you want text on action

---

14.3. The Grid object

> 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 class used to override defaults for styling your rendered grid. Allows you to specify classes or styles to apply at certain points in the grid
- icon_style: default: IconStyleFontawsome. used to get icon css classes. Other options: IconStyle, IconStyleBootstrapIcons
- T: optional pluralize object

(*) The parameters `details`, `editable` and `deletable` can also take a **callable** that will be passed the current row of the grid. This is useful because you can then turn a button on or off depending on the values in the row. In other words, instead of providing a simple Boolean value you can use an expression like:

```
deletable=lambda row: False if row.job=="CEO" else True,
```

See also Section 14.7.2 later on.

14.3.1 Searching and 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:

- build a search form. If more than one search query in the list, it will also generate a dropdown to select which search field to search against
- gather filter values and filter the grid

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.

14.3.2 CRUD settings

The grid provides CRUD (create, read, update and delete) capabilities utilizing py4web Form. You can turn off CRUD features by setting create/details/editable/deletable during grid 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.

14.4 Custom columns

If the grid does not involve a join but displays results from a single table you can specify a list of columns. Columns are highly customizable.

```
from py4web.utils.grid import Column
from yatl.helpers import A

columns = [
    db.person.id,
    db.person.superhero,
    db.person.name,
    db.person.job,
```

```
    Column("Web Site", lambda row: f"https://{row.superhero}.com"),
    Column("Go To", lambda row: A("link", _href=f"https://{row.superhero}.com"))
]

grid = Grid(... columns=columns ...)
```

Notice in this example the first columns are regular fields, The fifth column has a header "Web Site" and consists of URL strings generated from the rows. The last column has a header "Go To" and generates actual clickable links using the A helper. This is the result:
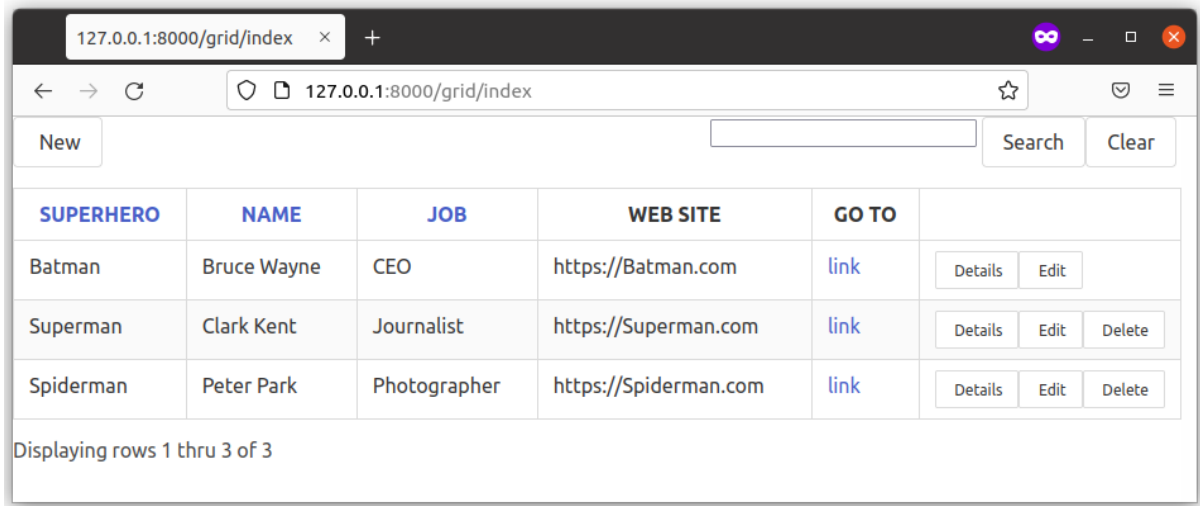
Notice that we've also used the `deletable` parameter in order to disable and hide it for Batman only, as explained before.

> **Warning** Do not define columns outside of the controller methods that use them, otherwise the structure of the table will change every time the user press the refresh button of the browser!
>
> The reason is that each time the grid displays, it modifies the 'columns' variable (in the grid) by adding the action buttons to it. So, if columns are defined outside of the controller method, it just keeps adding the actions column.

14.5 Using templates

Use the following to render your grid or CRUD forms in your templates.

Display the grid or a CRUD Form

```
[[=grid.render()]]
```

You can customize the CRUD form layout like a normal form (see Section 12.5.2). So you can use the following structure:

```
[[form = grid.render() ]]
[[form.custom["begin"] ]]
...
[[form.custom["submit"] ]]
[[form.custom["end"] ]]
```

But notice that when handling custom form layouts you need to know if you are displaying the grid or a form. Use the following to decide:

```
[[if grid.action in ['details', 'edit']: ]]
    #  Display the custom form
```

```
    [[form = grid.render() ]]
    [[form.custom["begin"] ]]
    ...
    [[form.custom["submit"]
    [[form.custom["end"]
[[else:]]
    [[grid.render() ]]
[[pass]]
```

14.6 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.
- icon_style is a class which provides the icon classes, for example for FontAwesome

The default `GridClassStyle` - based on **no.css**, primarily uses styles to modify the layout of the grid. We've already seen that it's possible to use other class_style, in particular `GridClassStyleBulma` or `GridClassStyleBootstrap5`.

You can even build your own class_style to be used with the css framework of your choice.

With icon_style, you can customize the icon font used. Currently, the following exist:

- `IconStyleFontawsome` is the current default (for backwards compatibility). You need to add fontawesome icon font CSS to use this.
- `IconStyle` which doesn't correspond to any font or icon set. It inserts simple css classes like `icon-edit-button` which you can write your own css for.
- `IconStyleBootstrapIcons` You need to add the icon font CSS from Bootstrap Icons to your html templates to use this.

14.7 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

You can build your own Action Button class to pass to pre/post action buttons based on the template below (this is not provided with py4web).

14.7.1 Sample Action Button Class

```
class GridActionButton:
  def __init__(
      self,
      url,
      text=None,
      icon=None,
```

```
        additional_classes="",
        additional_styles="",
        override_classes="",
        override_styles="",
        message="",
        append_id=False,
        name=None,
        ignore_attribute_plugin=False,
        **attrs
):
    self.url = url
    self.text = text
    self.icon = icon
    self.additional_classes = additional_classes
    self.additional_styles = additional_styles
    self.override_classes = override_classes
    self.override_styles = override_styles
    self.message = message
    self.append_id = append_id
    self.name = name
    self.ignore_attribute_plugin = ignore_attribute_plugin
    self.attrs = attrs
```

- url: the page to navigate to when the button is clicked

- text: text to display on the button

- icon: the icon corresponding to your icon-style to display before the text, for example "fa-calendar" for IconStyleFontawesome.

- additional_classes: a space-separated list of classes to include on the button element

- additional_styles: a string containing additional styles to add to the button

- override_classes: a space-separated list of classes to place on the control that will replace the default classes

- override_styles: a string containing the styles to be applied to the control

- 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

- name: the name to apply to the control

- ignore_attribute_plugin: boolean - respect the attribute plugin specified on the grid or ignore it

- attrs: additional attributes to apply to the control

After defining the custom GridActionButton class, you need to define your Action buttons:

```
pre_action_buttons = [
    lambda row: GridActionButton(
        lambda row: f"https://www.google.com/search?q={row.superhero}",
        text= f"Google for {row.superhero}",
    )
]
```

Finally, you need to reference them in the Grid definition:

```
grid = Grid(... pre_action_buttons = pre_action_buttons  ...)
```

14.7.2 Using callable parameters

A recent improvement to py4web allows you to pass a **callable** instead of a GridActionButton. This allow you to more easily change the behaviour of standard and custom Actions.

Callable can be used with:

- details

- editable

- deletable

- additional_classes

- additional_styles

- override_classes

- override_styles

Example usage:

```
@action("example")
def example():

    pre_action_buttons = [
        lambda row: GridActionButton(
            URL("test", row.id),
            text="Click me",
            icon=IconStyleFontawsome.add_button, # same as "fa-plus"
            additional_classes=row.id,
            additional_styles=["height: 10px" if row.bar else None],
        )
    ]

    post_action_buttons = [
        lambda row: GridActionButton(
            URL("test", row.id),
            text="Click me!!!",
            icon="fa-plus",
            additional_classes=row.id,
            additional_styles=["height: 10px" if row.bar else None],
        )
    ]

    grid = Grid(
        query=db.foo,
        pre_action_buttons=pre_action_buttons,
        post_action_buttons=post_action_buttons,
    )

    return dict(grid=grid.render())
```

14.8 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.

Also you can 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.

14.9 Grids with checkboxes

While the grid, per se, does not support checkboxes, you can use custom columns to add one or more columns of checkboxes. You can also add the helpers logic (the grid uses helpers to generate HTML) to wrap it in a `<form>` and add one or more submit buttons. You can then add logic to process the selected rows when the button is selected. For example:

```python
column = Column("select", lambda row:
INPUT(_type="checkbox",_name="selected_id",_value=row.id))

@action("manage")
@action.uses("manage.html", db)
def manage():

    grid = Grid(db.thing, columns=[column, db.thing.name])

    # if we are displaying a "select" grid page (not a form)
    if not grid.form:
        grid = grid.render()
        # if checkboxes selection was submitted
        if request.method == "POST":
            # do something with the selected ids
            print("you selected", request.POST.get("selected_id"))
        # inject a ``<form>`` and a ``submit`` button
        grid.children[1:] = [FORM(
                *grid.children[1:],
                DIV(INPUT(_type="submit",_value="do it!")),
                _method="POST",
                _action=request.url)]
    return locals()
```

Notice in the above example `request.POST.get("selected_id")` can be a single ID (if one selected) or a list of IDs (if more than one is selected).

From web2py to py4web

This chapter is dedicated to help users for porting old web2py applications to py4web.

Web2py and py4web share many similarities and some differences. For example they share the same database abstraction layer (pyDAL) which means pydal table definitions and queries are identical between the two frameworks. They also share the same template language with the minor caveat that web2py defaults to *{{…}}* delimiters while py4web defaults to *[[…]]* delimiters. They also share the same validators, part of pyDAL, and very similar helpers. The py4web ones are a lighter/-faster/minimalist re-implementation but they serve the same purpose and support a very similar syntax. They both provide a *Form* object (equivalent to *SQLFORM* in web2py) and a *Grid* object (equivalent to *SQLFORM.grid* in web2py). They both provide a *XML* object that can sanitize HTML and *URL* helper to generate URL. They both can raise *HTTP* to return non-200 OK pages. They both provide an *Auth* object that can generate register/login/change password/lost password/edit profile forms. Both web2py and py4web track and log all errors.

Some of the main differences are the following:

- web2py works with both Python 2.6+ and 3.6+, while py4web runs on Python 3.7+ only. So, if your old web2py application is still using Python 2, your first step involves migrating it to at least Python 3.7, better if the latest 3.9.

- web2py apps consist of collection of files which are executed at every HTTP request (using a custom importer, in a predetermined order). In py4web apps are regular python modules that are imported automatically by the frameworks. By the way, this makes possible the use of standard python debuggers (even inside the most used IDEs).

- In web2py every app has a fixed folder structure. A function is an action if and only if it is defined in a `controllers/*.py` file. py4web is much less constraining. In py4web an app must have an entry point `__init__.py` and a `static` folder. Every other convention such as the location of templates, uploaded files, translation files, sessions, etc. is user specified.

- In web2py the scaffolding app (the blue print for creating new apps) is called "welcome". In py4web it is called "_scaffold". _scaffold contains a "settings.py" file and a "common.py". The latter provides an example of how to enable Auth and configure all the options for the specific app. _scaffold has also a "model.py" file and a "controller.py" file but, unlike web2py, those files are not treated in any special manner. Their names follow a convention (not enforced by the framework) and they are imported by the *__init__.py* file as for any regular python module.

- In web2py every function in `controllers/*.py` is an action. In py4web a function is an action if it has the `@action("...")` decorator. That means that actions can be defined anywhere. The admin interface will help you locate where a particular action is defined.

- In web2py the mapping between URLs and file/function names is automatic but it can be over-written in "routes.py" (like in Django). In py4web the mapping is specified in the decorator as in *@action('my_url_path')* (like in Bottle and Flask). Notice that if the path starts with "/" it is assumed to be an absolute path. If not, it is assumed to be relative and prepended by the "/{app-name}/" prefix. Also, if the path ends with "/index", the latter postfix is assumed to be optional.

- In web2py the path extension matters and "http://.html" *is expected to return HTML while* "*http://*.json" is expected to return JSON, etc. In py4web there is no such convention. If the action returns a dict() and has a template, the dict() will be rendered by the template, else it will be rendered in JSON. More complex behavior can be accomplished using decorators.

- In web2py there are many wrappers around each action and, for example, they could handle sessions, pluralization, database connections, and more whether the action needs it or not. This makes web2py performances hard to compare with other frameworks. In py4web everything is optional and features must be enabled and configured for each action using the `@action.uses(...)` decorator. The arguments of `@action.uses(...)` are called fixtures in analogy with the fixtures in a house. They add functionality by providing preprocessing and postprocessing to the action. For example `@action.uses(session, T, db, flash)` indicates that the action needs to use session, internationalization/pluralization (T), the database (db), and carry on state for flash messages upon redirection.

- web2py uses its own request/response objects. py4web uses the request/response objects from the underlying Ombott library. While this may change in the future we are committed to keep them the interface with the web server, routing, partial requests, if modified since, and file streaming.

- Both web2py and py4web use the same pyDAL therefore tables are defined using the same exact syntax, and so do queries. In web2py tables are re-defined at every HTTP request, when the entire models are executed. In py4web only the action is executed for every HTTP request, while the code defined outside of actions is only executed at startup. That makes py4web much faster, in particular when there are many tables. The downside of this approach is that the developer should be careful to never override pyDAL variables inside action or in any way that depends on the content of the request object, else the code is not thread safe. The only variables that can be changed at will are the following field attributes: readable, writable, requires, update, default. All the others are for practical purposes to be considered global and non thread safe. This is also the reason that makes using Section 7.3.5 with py4web useless and even dangerous.

- Both web2py and pyweb have an Auth object which serve the same purpose. Both objects have the ability to generate forms pretty much in the same manner. The py4web ones is defined to be more modular and extensible and support both Forms and APIs, but it lacks the *auth.requires_** decorators and group membership/permissions. This does not mean that the feature is not available. In fact py4web is even more powerful and that is why the syntax is different. While the web2py Auth objects tries to do everything, the corresponding py4web object is only in charge of establishing the identity of a user, not what the user can do. The latter can be achieved by attaching Tags to users. So group membership is assigned by labeling users with the Tags of the groups they belong to and checking permissions based on the user tags. Py4web provides a mechanism for assigning and checking tags efficiently to any object, including but not limited to, users.

- Web2py comes with the Rocket web server. py4web at the time of writing defaults to the Rocket3 web server, which is the same multi-threaded web server used by web2py stripped of all the Python2 logic and dependencies. Note that this may change in the future.

15.1 Simple conversion examples

15.1.1 "Hello world" example

web2py

```python
# in controllers/default.py
def index():
    return "hello world"
```

–> **py4web**

```
# file imported by __init__.py
@action('index')
def index():
    return "hello world"
```

15.1.2 "Redirect with variables" example

web2py

```
request.get_vars.name
request.post_vars.name
request.env.name
raise HTTP(301)
redirect(url)
URL('c','f',args=[1,2],vars={})
```

–> **py4web**

```
request.query.get('name')
request.forms.get('name') or request.json.get('name')
request.environ.get('name')
raise HTTP(301)
redirect(url)
URL('c', 'f', 1, 2, vars={})
```

15.1.3 "Returning variables" example

web2py

```
def index():
    a = request.get_vars.a
    return locals()
```

–> **py4web**

```
@action("index")
def index():
    a = request.query.get('a')
    return locals()
```

15.1.4 "Returning args" example

web2py

```
def index():
    a, b, c = request.args
    b, c = int(b), int(c)
    return locals()
```

–> **py4web**

```
@action("index/<a>/<b:int>/<c:int>")
def index(a,b,c):
    return locals()
```

15.1.5 "Return calling methods" example

web2py

```
def index():
    if request.method == "GET":
```

```
        return "GET"
    if request.method == "POST":
        return "POST"
    raise HTTP(400)
```

–> py4web

```python
@action("index", method="GET")
def index():
    return "GET"


@action("index", method="POST")
def index():
    return "POST"
```

15.1.6 "Setting up a counter" example

web2py

```python
def counter():
    session.counter = (session.counter or 0) + 1
    return str(session.counter)
```

–> py4web

```python
def counter():
    session['counter'] = session.get('counter', 0) + 1
    return str(session['counter'])
```

15.1.7 "View" example

web2py

```
{{ extend 'layout.html' }}
<div>
{{ for k in range(1): }}
<span>{{= k }}<span>
{{ pass }}
</div>
```

–> py4web

```
[[ extend 'layout.html' ]]
<div>
[[ for k in range(1): ]]
<span>[[= k ]]<span>
[[ pass ]]
</div>
```

15.1.8 "Form and flash" example

web2py

```python
db.define_table('thing', Field('name'))

def index():
    form = SQLFORM(db.thing)
    form.process()
    if form.accepted:
        flash = 'Done!'
    rows = db(db.thing).select()
    return locals()
```

–> **py4web**

```python
db.define_table('thing', Field('name'))

@action("index")
@action.uses(db, flash)
def index():
    form = Form(db.thing)
    if form.accepted:
        flash.set("Done!", "green")
    rows = db(db.thing).select()
    return locals()
```

In the template you can access the flash object with

```html
<div class="flash">[[=globals().get('flash','')]]</div>
```

or using the more sophisticated

```html
<flash-alerts class="padded " data-alert="[[=globals().get( 'flash',
'')]]"></flash-alerts>
```

The latter requires `utils.js` from the scaffolding app to render the custom tag into a div with dismissal behavior.

Also notice that `Flash` is special: it is a singleton. So if you instantiate multiple Flash objects they share their data.

15.1.9 "grid" example

web2py

```python
def index():
    grid = SQLFORM.grid(db.thing, editable=True)
    return locals()
```

–> **py4web**

```python
@action("index")
@action.uses(db, flash)
def index():
    grid = Grid(db.thing)
    form.param.editable = True
    return locals()
```

15.1.10 "Accessing OS files" example

web2py

```python
file_path = os.path.join(request.folder, 'file.csv')
```

–> **py4web**

```python
from .settings import APP_FOLDER
file_path = os.path.join(APP_FOLDER, 'file.csv')
```

15.1.11 "auth" example

web2py

```python
auth = Auth()
auth.define_tables()
```

```python
@requires_login()
def index():
    user_id = auth.user.id
    user_email = auth.user.email
    return locals()


def user():
    return dict(form=auth())
```

Access with `http://.../user/login`.

–> **py4web**

```python
auth = Auth(define_table=False)
auth.define_tables()
auth.enable(route='auth')

@action("index")
@action.uses(auth.user)
def index():
    user_id = auth.user_id
    user_email = auth.get_user().get('email')
    return locals()
```

Access with `http://.../auth/login`. Notice that in web2py `auth.user` is the current logged-in user retrieved from session. In py4web instead `auth.user` is a fixture which serves the same purpose as `@requires_login` in web2py. In py4web only the `user_id` is stored in the session and it can be retrieved using `auth.user_id`. If you need more information about the user, you need to fetch the record from the database with `auth.get_user()` The latter returns all readable fields as a Python dictionary.

Also notice there is a big difference between:

```python
@action.uses(auth)
```

and

```python
@action.uses(auth.user)
```

In the first case the decorated action can access the auth object but `auth.user_id` may be None if the user is not logged in. In the second case we are requiring a valid logged in user and therefore `auth.user_id` is guaranteed to be a valid user id.

Also notice that if an action uses auth, then it automatically uses its session and its flash objects.

Advanced topics and examples

16.1 The scheduler

Py4web has a built-in scheduler. There is nothing for you to install or configure to make it work.

Given a task (just a python function), you can schedule async runs of that function. The runs can be a one-off or periodic. They can have timeout. They can be scheduled to run at a given scheduled time.

The scheduler works by creating a table `task_run` and enqueueing runs of the predefined task as table records. Each `task_run` references a task and contains the input to be passed to that task. The scheduler will capture the task stdout+stderr in a `db.task_run.log` and the task output in `db.task_run.output`.

A py4web thread loops and finds the next task that needs to be executed. For each task it creates a worker process and assigns the task to the worker process. You can specify how many worker processes should run concurrently. The worker processes are daemons and they only live for the life of one task run. Each worker process is only responsible for executing that one task in isolation. The main loop is responsible for assigning tasks and timeouts.

The system is very robust because the only source of truth is the database and its integrity is guaranteed by transactional safety. Even if py4web is killed, running tasks continue to run unless they complete, fail, or are explicitly killed.

Aside for allowing multiple concurrent task runs in execution on one node, it is also possible to run multiple instances of the scheduler on different computing nodes, as long as they use the same client/server database for `task_run` and as long as they all define the same tasks.

Here is an example of how to use the scheduler:

```python
from pydal.tools.scheduler import Scheduler, delta, now
from .common import db

# create and start the scheduler
scheduler = Scheduler(db, sleep_time=1, max_concurrent_runs=1)
scheduler.start()

# register your tasks
scheduler.register_task("hello", lambda **inputs: print("hi!"))
scheduler.register_task("slow", lambda: time.sleep(10))
scheduler.register_task("periodic", lambda **inputs: print("I am periodic!"))
scheduler.register_task("fail", lambda x: 1 / x)

# enqueue some task runs:

scheduler.enqueue_run(name="hello")
scheduler.enqueue_run(name="hello", scheduled_for=now() + delta(10) # start in 10
secs
```

```
scheduler.enqueue_run(name="slow", timeout=1) # 1 secs
scheduler.enqueue_run(name="periodic", period=10) # 10 secs
scheduler.enqueue_run(name="fail", inputs={"x": 0})
```

Notice that in scaffolding app, the scheduler is created and started in common if `USE_SCHEDULER=True` in `settings.py`.

You can manage your task runs busing the dashboard or using a `Grid(db.task_run)`.

To prevent database locks (in particular with sqlite) we recommend:

- Use a different database for the scheduler and everything else

- Always `db.commit()` as soon as possible after any insert/update/delete

- wrap your database logic in tasks in a try…except as in

```python
def my_task():
    try:
        # do something
        db.commit()
    except Exception:
        db.rollback()
```

16.2 Sending messages using a background task

As en example of application of the above, consider the case of wanting to send emails asynchronously from a background task. In this example we send them using SendGrid from Twilio (https://www.twilio.com/docs/sendgrid/for-developers/sending-email/quickstart-python).

Here is a possible scheduler task to send the email:

```python
import sendgrid
from sendgrid.helpers.mail import Mail, Email, To, Content

def sendmail_task(from_addr, to_addrs, subject, body):
    ""
    # build the messages using sendgrid API
    from_email = Email(from_addr)  # Must be your verified sender
    content_type = "text/plain" if body[:6] != "<html>" else "text/html"
    content = Content(content_type, body)
    mail = Mail(from_email, To(to_addrs), subject, content)
    # ask sendgrid to deliver it
    sg = sendgrid.SendGridAPIClient(api_key=settings.SENDGRID_API_KEY)
    response = sg.client.mail.send.post(request_body=mail.get())
    # check if worked
    assert response.status_code == "200"

# register the above task with the scheduler
scheduler.register_task("sendmail", sendmail_task)
```

To schedule sending a new email do:

```python
email = {
    "from_addr": "me@example.com",
    "to_addrs": ["me@example.com"],
    "subject": "Hello World",
    "body": "I am alive!",
}
scheduler.enqueue_run(name="sendmail", inputs=email, scheduled_for=None)
```

The key:value in the email representation must match the arguments of the task. The `scheduled_for` argument is optional and allows you to specify when the email should be sent.

You can use the Dashboard to see the status of your `task_runs` for the task called `sendmail`.

You can also tell auth to tap into above mechanism for sending emails:

```python
class MySendGridSender:
    def __init__(self, from_addr):
        self.from_addr = from_adds
    def send(self, to_addr, subject, body):
        email = {
            "from_addr": self.from_addr,
            "to_addrs": [to_addr],
            "subject": subject,
            "body": body,
        }
        scheduler.enqueue_run(name="sendmail", inputs=email)

auth.sender = MySendGridSender(from_addr="me@example.com")
```

With the above, Auth will not send emails using smtplib. Instead it will send them with SendGrid using the scheduler. Notice the only requirement here is that `auth.sender` must be an object with a `send` method with the same signature as in the example.

Notice, it it also possible to send SMS messages instead of emails but this requires 1) store the phone number in `auth_user` and 2) override the `Auth.send` method.

16.3 Celery

Yes. You can use Celery instead of the build-in scheduler but it adds complexity and it is less robust. Yet the build-in scheduler is designed for long running tasks and the database can become a bottleneck if you have hundreds of tasks running concurrently. Celery may work better if you have more than 100 concurrent tasks and/or they are short running tasks.

16.4 py4web and asyncio

Asyncio is not strictly needed, at least for most of the normal use cases where it will add problems more than value because of its concurrency model. On the other hand, we think py4web needs a built-in websocket async based solution.

If you plan to play with asyncio be careful that you should also deal with all the framework's components: in particular pydal is not asyncio compliant because not all the adapters work with async.

16.5 htmx

There are many javascript front-end frameworks available today that allow you great flexibility over how you design your web client. Vue, React and Angular are just a few. However, the complexity in building one of these systems prevents many developers from reaping those benefits. Add to that the rapid state of change in the ecosystem and you soon have an application that is difficult to maintain just a year or two down the road.

As a consequence, there is a growing need to use simple html elements to add reactivity to your web pages. htmx is one of the tools emerging as a leader in page reactivity without the complexities of javascript. Technically, htmx allows you to access AJAX, CSS Transitions, Web Sockets and Server Sent Events directly in HTML, using attributes, so you can build modern user interfaces with the simplicity and power of hypertext. [CIT1601]

Read all about htmx and its capabilities on the official site at https://htmx.org . If you prefer, there is also a video tutorial: Simple, Fast Frontends With htmx .

[CIT1601] from the https://htmx.org website

---

py4web enables htmx integration in a couple of ways.

1. Allow you to add htmx attributes to your forms and buttons

2. Includes an htmx attributes plugin for the py4web grid

16.5.1 htmx usage in Form

The py4web Form class allows you to pass **kwargs to it that will be passed along as attributes to the html form. For example, to add the hx-post and hx-target to the <form> element you would use:

```
attrs = {
    "_hx-post": URL("url_to_post_to/%s" % record_id),
    "_hx-target": "#detail-target",
}
form = Form(
    db.tablename,
    record=record_id,
    **attrs,
)
```

Now when your form is submitted it will call the URL in the hx-post attribute and whatever is returned to the browser will replace the html inside of the element with id="detail-target".

Let's continue with a full example (started from scaffold).

controllers.py

```
import datetime

@action("htmx_form_demo", method=["GET", "POST"])
@action.uses("htmx_form_demo.html")
def htmx_form_demo():
    return dict(timestamp=datetime.datetime.now())


@action("htmx_list", method=["GET", "POST"])
@action.uses("htmx_list.html", db)
def htmx_list():
    superheros = db(db.superhero.id > 0).select()
    return dict(superheros=superheros)


@action("htmx_form/<record_id>", method=["GET", "POST"])
@action.uses("htmx_form.html", db)
def htmx_form(record_id=None):
    attrs = {
        "_hx-post": URL("htmx_form/%s" % record_id),
        "_hx-target": "#htmx-form-demo",
    }
    form = Form(db.superhero, record=db.superhero(record_id), **attrs)
    if form.accepted:
        redirect(URL("htmx_list"))

    cancel_attrs = {
        "_hx-get": URL("htmx_list"),
        "_hx-target": "#htmx-form-demo",
    }
    form.param.sidecar.append(A("Cancel", **cancel_attrs))

    return dict(form=form)
```

templates/htmx_form_demo.html

```
[[extend 'layout.html']]

[[=timestamp]]
<div id="htmx-form-demo">
    <div hx-get="[[=URL('htmx_list')]]" hx-trigger="load"
hx-target="#htmx-form-demo"></div>
</div>


<script src="https://unpkg.com/htmx.org@1.3.2"></script>
```

templates/htmx_list.html

```
<ul>
[[for sh in superheros:]]
    <li><a hx-get="[[=URL('htmx_form/%s' % sh.id)]]"
hx-target="#htmx-form-demo">[[=sh.name]]</a></li>
[[pass]]
</ul>
```

templates/htmx_form.html

```
[[=form]]
```

We now have a functional maintenance app to update our superheros. In your browser navigate to the htmx_form_demo page in your new application. The hx-trigger="load" attribute on the inner div of the htmx_form_demo.html page loads the htmx_list.html page inside the htmx-form-demo DIV once the htmx_form_demo page is loaded.

Notice the timestamp added outside of the htmx-form-demo DIV does not change when transitions occur. This is because the outer page is never reloaded, only the content inside the htmx-form-demo DIV.

The htmx attributes hx-get and hx-target are then used on the anchor tags to call the htmx_form page to load the form inside the htmx-form-demo DIV.

So far we've just seen standard htmx processing. Nothing fancy here, and nothing specific to py4web. However, in the htmx_form method we see how you can pass any attribute to a py4web form that will be rendered on the <form> element as we add the hx-post and hx-target. This tells the form to allow htmx to override the default form behavior and to render the resulting output in the target specified.

The default py4web form does not include a Cancel button in case you want to cancel out of the edit form. But you can add 'sidecar' elements to your forms. You can see in htmx_form that we add a cancel option and add the required htmx attributes to make sure the htmx_list page is rendered inside the htmx-form-demo DIV.

16.5.2 htmx usage in Grid

The py4web grid provides an attributes plugin system that allows you to build plugins to provide custom attributes for form elements, anchor elements or confirmation messages. py4web also provide an attributes plugin specifically for htmx.

Here is an example building off the previous htmx forms example.

controller.py

```
@action("htmx_form/<record_id>", method=["GET", "POST"])
@action.uses("htmx_form.html", db)
def htmx_form(record_id=None):
    attrs = {
        "_hx-post": URL("htmx_form/%s" % record_id),
        "_hx-target": "#htmx-form-demo",
    }
    form = Form(db.superhero, record=db.superhero(record_id), **attrs)
    if form.accepted:
```

```
        redirect(URL("htmx_list"))

    cancel_attrs = {
        "_hx-get": URL("htmx_list"),
        "_hx-target": "#htmx-form-demo",
    }
    form.param.sidecar.append(A("Cancel", **cancel_attrs))

    return dict(form=form)

@action("htmx_grid")
@action.uses( "htmx_grid.html", session, db)
def htmx_grid():
    grid = Grid(db.superhero, auto_process=False)

    grid.attributes_plugin = AttributesPluginHtmx("#htmx-grid-demo")
    attrs = {
        "_hx-get": URL(
            "htmx_grid",
        ),
        "_hx-target": "#htmx-grid-demo",
    }
    grid.param.new_sidecar = A("Cancel", **attrs)
    grid.param.edit_sidecar = A("Cancel", **attrs)

    grid.process()

    return dict(grid=grid)
```

templates/htmx_form_demo.html

```
[[extend 'layout.html']]

[[=timestamp]]
<div id="htmx-form-demo">
    <div hx-get="[[=URL('htmx_list')]]" hx-trigger="load"
hx-target="#htmx-form-demo"></div>
</div>

<div id="htmx-grid-demo">
    <div hx-get="[[=URL('htmx_grid')]]" hx-trigger="load"
hx-target="#htmx-grid-demo"></div>
</div>

<script src="https://unpkg.com/htmx.org@1.3.2"></script>
```

Notice that we added the #htmx-grid-demo DIV which calls the htmx_grid route.

templates/htmx_grid.html

```
[[=grid.render()]]
```

In htmx_grid we take advantage of deferred processing on the grid. We setup a standard CRUD grid, defer processing and then tell the grid we're going to use an alternate attributes plugin to build our navigation. Now the forms, links and delete confirmations are all handled by htmx.

16.5.3 Autocomplete Widget using htmx

htmx can be used for much more than just form/grid processing. In this example we'll take advantage of htmx and the py4web form widgets to build an autocomplete widget that can be used in your forms. *NOTE: this is just an example, none of this code comes with py4web*

Again we'll use the superheros database as defined in the examples app.

Add the following to your controllers.py. This code will build your autocomplete dropdowns as well

as handle the database calls to get your data.

```python
import json
from functools import reduce

from yatl import DIV, INPUT, SCRIPT

from py4web import action, request, URL
from ..common import session, db, auth


@action(
    "htmx/autocomplete",
    method=["GET", "POST"],
)
@action.uses(
    "htmx/autocomplete.html",
    session,
    db,
    auth.user,
)
def autocomplete():
    tablename = request.params.tablename
    fieldname = request.params.fieldname
    autocomplete_query = request.params.query

    field = db[tablename][fieldname]
    data = []

    fk_table = None

    if field and field.requires:
        fk_table = field.requires.ktable
        fk_field = field.requires.kfield

        queries = []
        if "_autocomplete_search_fields" in dir(field):
            for sf in field._autocomplete_search_fields:
                queries.append(
                    db[fk_table][sf].contains(
                        request.params[f"{tablename}_{fieldname}_search"]
                    )
                )
            query = reduce(lambda a, b: (a | b), queries)
        else:
            for f in db[fk_table]:
                if f.type in ["string", "text"]:
                    queries.append(
                        db[fk_table][f.name].contains(
                            request.params[f"{tablename}_{fieldname}_search"]
                        )
                    )

            query = reduce(lambda a, b: (a | b), queries)

        if len(queries) == 0:
            queries = [db[fk_table].id > 0]
            query = reduce(lambda a, b: (a & b), queries)

        if autocomplete_query:
            query = reduce(lambda a, b: (a & b), [autocomplete_query, query])
        data = db(query).select(orderby=field.requires.orderby)
```

```python
    return dict(
        data=data,
        tablename=tablename,
        fieldname=fieldname,
        fk_table=fk_table,
        data_label=field.requires.label,
    )


class HtmxAutocompleteWidget:
    def __init__(self, simple_query=None, url=None, **attrs):
        self.query = simple_query
        self.url = url if url else URL("htmx/autocomplete")
        self.attrs = attrs

        self.attrs.pop("simple_query", None)
        self.attrs.pop("url", None)

    def make(self, field, value, error, title, placeholder="", readonly=False):
        # TODO: handle readonly parameter
        control = DIV()
        if "_table" in dir(field):
            tablename = field._table
        else:
            tablename = "no_table"

        # build the div-hidden input field to hold the value
        hidden_input = INPUT(
            _type="text",
            _id="%s_%s" % (tablename, field.name),
            _name=field.name,
            _value=value,
        )
        hidden_div = DIV(hidden_input, _style="display: none;")
        control.append(hidden_div)

        # build the input field to accept the text

        # set the htmx attributes

        values = {
            "tablename": str(tablename),
            "fieldname": field.name,
            "query": str(self.query) if self.query else "",
            **self.attrs,
        }
        attrs = {
            "_hx-post": self.url,
            "_hx-trigger": "keyup changed delay:500ms",
            "_hx-target": "#%s_%s_autocomplete_results" % (tablename, field.name),
            "_hx-indicator": ".htmx-indicator",
            "_hx-vals": json.dumps(values),
        }
        search_value = None
        if value and field.requires:
            row = (
                db(db[field.requires.ktable][field.requires.kfield] == value)
                .select()
                .first()
            )
            if row:
                search_value = field.requires.label % row

        control.append(
```

```
            INPUT(
                _type="text",
                _id="%s_%s_search" % (tablename, field.name),
                _name="%s_%s_search" % (tablename, field.name),
                _value=search_value,
                _class="input",
                _placeholder=placeholder if placeholder and placeholder != "" else
"..",
                _title=title,
                _autocomplete="off",
                **attrs,
            )
        )

        control.append(DIV(_id="%s_%s_autocomplete_results" % (tablename,
field.name)))

        control.append(
            SCRIPT(
                """
        htmx.onLoad(function(elt) {
            document.querySelector('#%(table)s_%(field)s_search').onkeydown =
check_%(table)s_%(field)s_down_key;
            \n
            function check_%(table)s_%(field)s_down_key(e) {
                if (e.keyCode == '40') {

document.querySelector('#%(table)s_%(field)s_autocomplete').focus();

document.querySelector('#%(table)s_%(field)s_autocomplete').selectedIndex = 0;
                }
            }
        })
        """
                % {
                    "table": tablename,
                    "field": field.name,
                }
            )
        )

        return control
```

Usage - in your controller code, this example uses bulma as the base css formatter.

```
formstyle = FormStyleFactory()
formstyle.classes = FormStyleBulma.classes
formstyle.class_inner_exceptions = FormStyleBulma.class_inner_exceptions
formstyle.widgets["vendor"] = HtmxAutocompleteWidget(
    simple_query=(db.vendor.vendor_type == "S")
)

form = Form(
    db.product,
    record=product_record,  # defined earlier in controller
    formstyle=formstyle,
)
```

First, get an instance of FormStyleFactory. Then get the base css classes from whichever css framework you wish. Add the class inner exceptions from your css framework. Once this is set up you can override the default widget for a field based on its name. In this case we're overriding the widget for the 'vendor' field. Instead of including all vendors in the select dropdown, we're limiting only to those with a vendor type equal to 'S'.

When this is rendered in your page, the default widget for the vendor field is replaced with the widget generated by the HtmxAutocompleteWidget. When you pass a simple query to the Htmx-AutocompleteWidget the widget will use the default route to fill the dropdown with data.

If using the simple query and default build url, you are limited to a simple DAL query. You cannot use DAL subqueries within this simple query. If the data for the dropdown requires a more complex DAL query you can override the default data builder URL to provide your own controller function to retrieve the data.

16.6 utils.js

Multiple times in this documentation we have mentioned utils.js which comes with the scaffolding application, yet we never clearly listed what is in there. So here it is.

16.6.1 string.format

It extends the String object prototype to allow expressions like this:

```
var a = "hello {name}".format(name="Max");
```

16.6.2 The Q object

The Q object can be used like a selector supporting jQuery like syntax:

```
var element = Q("#element-id")[0];
var selected_elements = Q(".element-class");
```

It supports the same syntax as JS `querySelectorAll` and always returns an array of selected elements (can be empty).

The Q objects is also a container for functions that can be useful when programming in Javascript. It is stateless.

For example:

Q.clone

A function to clone any object:

```
var b = {any: "object"}
var a = Q.clone(b);
```

Q.eval

It evaluates JS expressions in a string. It is not a sandbox.

```
var a = Q.eval("2+3+Math.random()");
```

Q.ajax

A wrapper for the JS fetch method which provides a nicer syntax:

```
var data = {};
var headers = {'custom-header-name': 'value'}
var success = response => { console.log("recereived", response); }
var failure = response => { console.log("recereived", response); }
Q.ajax("POST", url, data, headers).then(success, failure);
```

Q.get_cookie

Extracts a cookie by name from the header of cookies in the current page: returns null if the cookie does not exist. Can be used within the JS of a page to retrieve a session cookie in case it is needed to call an API.

```
var a = Q.get_cookie("session");
```

Q.register_vue_component

This is specific for Vue 2 and may be deprecated in the future but it allows to define a vue component where the template is stored in a separate HTML file and the template will be loaded lazily only when/if the component is used.

For example instead of doing:

```
Vue.component('button-counter', {
data: function () {
    return {
        count: 0
    }
},
template: '<button v-on:click="count++">You clicked me {{ count }}
times.</button>'
});
```

You would put the template in a button-counter.html and do

```
Q.register_vue_component("button-counter", "button-counter.html", function(res) {
    return {
        data: function () {
            return {
                count: 0
            };
        };
});
```

Q.upload_helper

It allows to bind an input tag of type file to a callback so that when a file is selected the content of the selected file is loaded, base64 encoded, and passed to the callback.

This is useful to create form which include an input field selector - but you want to place the content of the selected file into a variable, for example to do an ajax post of that content.

For example:

```
<input type="file" id="my-id" />
```

and

```
var file_name = ""
var file_content = "";
Q.upload_helper("my_id", function(name, content) {
    file_name = name;
    file_content = content; // base 64 encoded;
}
```

16.6.3 The T object

This is a Javascript reimplementation of the Python pluralize library in Python which is used by the Python T object in py4web. So basically a client-side T.

```
T.translations = {'dog': {0: 'no cane', 1: 'un case', 2: '{n} cani', 10: 'tanti
cani'}};
var message = T('dog').format({n: 5}); // "5 cani"
```

The intended usage is to create a server endpoint that can provide translations for the client accepted-language, obtain T.translations via ajax get, and then use T to translate and pluralize all messages clientside rather than serverside.

Q.debounce

Prevents a function from stepping on itself.

```
setInterval(500, Q.debounce(function(){console.log("hello!")}, 200);
```

and the function will be called every 500ms but will skip if the previous call did not terminate. Unlike other debounce implementations out there, it makes sure the last call is always executed by delaying it (in the example 200ms);

Q.debounce

Prevents a function from being called too often;

```
Q("#element").onclick = Q.debounce(function(){console.log("clicked!")}, 1000);
```

If the element is clicked more often than once every 1000ms, the other clicks will be ignored.

Q.tags_inputs

It turns a regular text input containing a string of comma separated tags into a tag widgets. For example:

```
<input name="browsers"/>
```

and in JSL

```
Q.tags_input('[name=zip_codes]')
```

You can restrict the set of options with:

```
Q.tags_input('[name=zip_codes]', {
   freetext: false,
   tags: ['Chrome', 'Firefox', 'Safari', 'Edge']
});
```

It works with the datalist element to provide autocomplete. Simply prepend -list to the datalist id:

```
<input name="browsers"/>
<datalist id="browses-list">
   <option>Chrome</option>
   <option>Firfox</option>
   <option>Safari</option>
   <option>Edge</option>
</datalist>
```

and in JS:

```
Q.tags_input('[name=zip_codes]', {freetext: false});
```

It provides more undocumented options. You need to style the tags. For example:

```
ul.tags-list {
  padding-left: 0;
}
ul.tags-list li {
  display: inline-block;
  border-radius: 100px;
  background-color: #111111;
  color: white;
  padding: 0.3em 0.8em 0.2em 0.8em;
  line-height: 1.2em;
  margin: 2px;
  cursor: pointer;
  opacity: 0.2;
  text-transform: capitalize;
```

```
}
ul.tags-list li[data-selected=true] {
  opacity: 1.0;
}
```

Notice that if an input element has class *.type-list-string* or *.type-list-integer*, utils.js applies the *tag_input* function automatically.

Q.score_input*

..code:: javascript

Q.score_input(Q('input[type=password]')[0]);

This will turn the password input into a widget that scores the password complexity. It is applied automatically to inputs with name "password" or "new_password".

Components

This is a poor man version of HTMX. It allows to insert in the page ajax-component tags that are loaded via ajax and any form in those components will be trapped (i.e. the result of form submission will also be displayed inside the same component)

For example imagine an index.html that contains

```
<ajax-component id="component_1" url="[[=URL('mycomponent')]]">
    <blink>Loading...</blink>
</ajax-component>
```

And a different action serving the component:

```
@action("mycomponent", method=["GET", "POST"])
@action.uses(flash)
def mycomponent():
    flash.set("Welcome")
    form = Form([Field("your_name")])
    return DIV(
        "Hello " + request.forms["your_name"]
        if form.accepted else form).xml()
```

A component action is a regular action except that it should generate html without the *<html><body>…</body></html>* envelop and it can make use of templates and flash for example.

Notice that if the main page supports flash messages, any flash message in the component will be displayed by the parent page.

Moreover if the component returns a *redirect("other_page")* not just the content of the component, but the entire page will be redirected.

The contents of the component html can contain *<script>…</script>* and they can modify global page variables as well as modify other components.

- genindex
- modindex
- search