



py4web Documentation

Versão 1.2025-preview

Massimo Di Pierro

1	O que é py4web?	1
1.1	Acknowledgments	2
2	Ajuda, recursos e dicas	5
2.1	Recursos	5
2.2	Dicas e sugestões	6
2.3	Como contribuir	7
3	Instalação e colocação em funcionamento	9
3.1	Understanding the design.	9
3.2	Plataformas e pré-requisitos suportados	9
3.3	Procedimentos de configuração	9
3.4	Melhoramento.	11
3.5	Primeira corrida.	12
3.6	Opções de linha de comando	13
3.7	Special installations.	17
4	O Dashboard	19
4.1	A página Web principal	19
4.2	Sessão no Dashboard	20
5	Creating an app	25
5.1	Do princípio	25
5.2	Páginas estáticas	25
5.3	Páginas web dinâmicas.	26
5.4	The _scaffold app	29
5.5	Copying the _scaffold app	31
5.6	Watch for files change	31
5.7	Domain-mapped apps	32
5.8	Custom error pages.	33
6	Fixtures	35
6.1	Using Fixtures.	35
6.2	The Template fixture	36
6.3	The Inject fixture	37
6.4	The Translator fixture.	37
6.5	O fixture flash	40
6.6	The Session fixture	41
6.7	The Condition fixture.	44
6.8	The URLsigner fixture	45
6.9	O fixture DAL	45

6.10	The Auth fixture	46
6.11	Caveats about fixtures	47
6.12	Fixtures personalizados	47
6.13	Multiple fixtures	50
6.14	Caching e Memoize.	50
6.15	Decoradores de conveniência	51
7	The Database Abstraction Layer (DAL)	53
7.1	DAL introduction	53
7.2	Using the dashboard app with databases	55
7.3	Construtor DAL	56
7.4	Construtor Table	62
7.5	Construtor Field.	65
7.6	Migrações	70
7.7	Table methods.	72
7.8	Raw SQL	77
7.9	`` Comando SELECT``	79
7.10	Computed and Virtual fields	90
7.11	Joins and Relations	93
7.12	Outros operadores	98
7.13	Exportar e importar dados	101
7.14	Características avançadas.	106
7.15	Pegadinhas	112
8	The RestAPI	117
8.1	RestAPI policies and actions	118
8.2	RestAPI GET	120
8.3	RestAPI practical examples.	121
8.4	The RestAPI response.	134
9	Linguagem de template YATL	135
9.1	Sintaxe básica	137
9.2	Information workflow.	139
9.3	Page layout standard structure.	143
10	Helpers YATL	145
10.1	Helpers overview	145
10.2	Built-in helpers.	147
10.3	Helpers personalizados.	152
10.4	Server-side <i>DOM</i>	153
10.5	Using Inject.	155
11	Internacionalização	157
11.1	Pluralizar	157
11.2	Atualizar os arquivos de tradução.	158
12	Formulários	159
12.1	The Form constructor	160
12.2	A minimal form example without a database	161
12.3	Basic form example	162
12.4	Widgets.	164
12.5	Advanced form design	166
12.6	Validação de formulário	167
13	Authentication and authorization	185
13.1	Authentication using Auth	185
13.2	Authorization using Tags.	193

14	Rede	197
14.1	Key features	197
14.2	Basic grid example	197
14.3	The Grid object.	200
14.4	Custom columns.	202
14.5	Usando templates	203
14.6	Customizing style	204
14.7	Ação personalizada Botões	204
14.8	Os campos de referência	206
14.9	Grids with checkboxes	207
15	De web2py para py4web	209
15.1	Simple conversion examples	210
16	Advanced topics and examples	215
16.1	The scheduler	215
16.2	Sending messages using a background task	216
16.3	Celery	217
16.4	py4web and asyncio.	217
16.5	htmx.	217
16.6	utils.js	224

O que é 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:

- Nós portado web2py para Python 3.
- Nós quebramos web2py em módulos que podem ser usados de forma independente.
- Nós reagrupados alguns desses módulos em uma nova e mais modular quadro ... 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, ao contrário web2py, requer 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, como web2py, podem servir múltiplas aplicações concorrentemente, enquanto as aplicações são submódulos do módulo de aplicações.
- 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, como web2py, tem um built-in sistema de bilhética, mas, ao contrário web2py, este sistema é global e não por aplicação. Os bilhetes já não são armazenados no sistema de arquivos com os aplicativos individuais. Eles são armazenados em um único banco de dados.
- PY4WEB, like web2py, is based on pydal but leverages some new features of pydal (RESTAPI).
- PY4WEB, como web2py, usa a linguagem de template yatl mas o padrão é suportes delimitadores quadrados para evitar conflitos com modelo quadros JS, como Vue.js e angularjs. Yatl inclui um subconjunto dos ajudantes web2py.
- PY4WEB, ao contrário web2py, usa a biblioteca pluralização para a internacionalização. Na prática, isso expõe um objeto T muito semelhante ao do web2py T mas fornece melhor cache e capacidades pluralização mais flexíveis.
- PY4WEB vem com um painel APP que administrador substitui do web2py. Esta é uma IDE web para carregar / gestão / aplicativos de edição.
- Painel de PY4WEB inclui uma interface de banco de dados baseado na web. Isto substitui a funcionalidade AppAdmin de web2py.
- PY4WEB comes with Form and Grid objects that are similar to web2py's SQLFORM and SQLFORM.grid.
- PY4WEB vem com um objeto Auth que substitui o web2py. É mais modular e mais fácil de estender. Fora da caixa, ele fornece a funcionalidade básica do registo, login, logout, de alteração de senha, solicitação de alteração de senha, editar o perfil, bem como a integração com o PAM, SAML2, LDAP, OAuth2 (google, facebook e Twitter).
- PY4WEB leverages PyDAL's new tags functionality to tag users with groups, search users by groups, and apply permissions based on membership.
- PY4WEB vem com alguns componentes personalizados Vue.js projetados para interagir com o PyDAL RESTAPI, e com PY4WEB em geral. Essas APIs são projetados para permitir que o servidor para definir políticas sobre quais operações um cliente é permitido para executar, mas dá a flexibilidade cliente dentro dessas restrições. Os dois principais componentes são mtable (que fornece uma interface baseada na web para o banco de dados semelhante à grade do web2py) e auth (uma interface personalizável à API Auth).

O objetivo do PY4WEB é e continua a ser o mesmo que web2py de: para o desenvolvimento web make fácil e acessível, enquanto a produção de aplicações que são rápidos e seguros.

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 axotl»: it magically represents the sense of kindness and inclusion. We believe it's the cornerstone of our growing community.



Ajuda, recursos e dicas

Nós fizemos o nosso melhor para tornar simples PY4WEB e limpo. Mas você sabe, moderno programação web é uma tarefa difícil. Ela exige uma mente aberta, capaz de saltar com frequência (sem ser perdida!) De python para HTML para javascript para css e gestão de banco de dados mesmo. Mas não tenha medo, neste manual vamos ajudá-lo lado a lado nesta jornada. E há muitos outros recursos valiosos que nós vamos mostrar-lhe.

2.1 Recursos

2.1.1 Este 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 O grupo Google

Existe uma lista de discussão dedicado hospedado no Google Groups, consulte <https://groups.google.com/g/py4web>. Esta é a principal fonte de discussões para desenvolvedores e usuários simples. Para qualquer problema que você deve enfrentar, este é o lugar certo para procurar uma dica ou uma solução.

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 Tutoriais e vídeo

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 As fontes no 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 Dicas e sugestões

Este parágrafo é dedicado a dicas preliminares, sugestões e dicas que podem ser úteis para saber antes de começar a aprender py4web.

2.2.1 Pré-requisitos

A fim de compreender py4web você precisa de pelo menos um conhecimento básico python. Há muitos livros, cursos e tutoriais disponíveis na Web - escolher o que é melhor para você. decoradores do Python, em particular, são um marco de qualquer quadro python web e você tem que compreendê-lo totalmente.

2.2.2 Um local de trabalho python moderna

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**.

Quando você vai começar a lidar com programas mais complexos e confiabilidade necessidade, sugerimos também para:

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

2.2.3 Depuração py4web com 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 Depuração py4web com PyCharm

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

2.3 Como contribuir

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.

Se você deseja corrigir e ampliar este manual, ou mesmo traduzi-lo em uma nova língua estrangeira, você pode ler todas as informações necessárias diretamente no `README` específica <https://github.com/web2py/py4web/blob/master/docs/README.md> no 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.

Instalação e colocação em funcionamento

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 Plataformas e pré-requisitos suportados

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 Procedimentos de configuração

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 instalação completa de qualquer aplicação python complexo como py4web certamente irá modificar o ambiente python do seu sistema. A fim de evitar qualquer alteração indesejada, é um bom hábito de usar um ambiente virtual python (também chamado **virtualenv**, veja [aqui <https://docs.python.org/3.7/tutorial/venv.html>](https://docs.python.org/3.7/tutorial/venv.html) para uma introdução). Este é um recurso padrão do Python; se você ainda não sabe virtualenv é um bom momento para começar a sua descoberta!

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`.

Isto irá instalar py4web e todas as suas dependências em único caminho do sistema. A pasta de ativos (que contém os aplicativos do sistema do py4web) também será criado. Após a instalação, você será capaz de começar a py4web em qualquer pasta de trabalho com

```
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 Instalação de fonte (globalmente)

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 Instalando a partir de binários

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.

A fim de usá-lo você só precisa fazer o download do arquivo mais recente do Windows ou MacOS zip do *este repositório externo* <<https://github.com/nicozanf/py4web-pyinstaller>> __. Descompacte-o em uma pasta local e abrir uma linha de comando lá. finalmente executar

```
./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 Melhoramento

Se você instalou `py4web` de pip você pode simples atualizá-lo com

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

Warning Isto não irá atualizar automaticamente os aplicativos padrão, como o Dashboard **** **** e padrão **** ****. Você tem que remover manualmente esses aplicativos e execute

```
py4web setup <path to apps_folder>
```

a fim de re-instalá-los. Esta é uma precaução de segurança, no caso de você fez alterações para esses aplicativos.

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

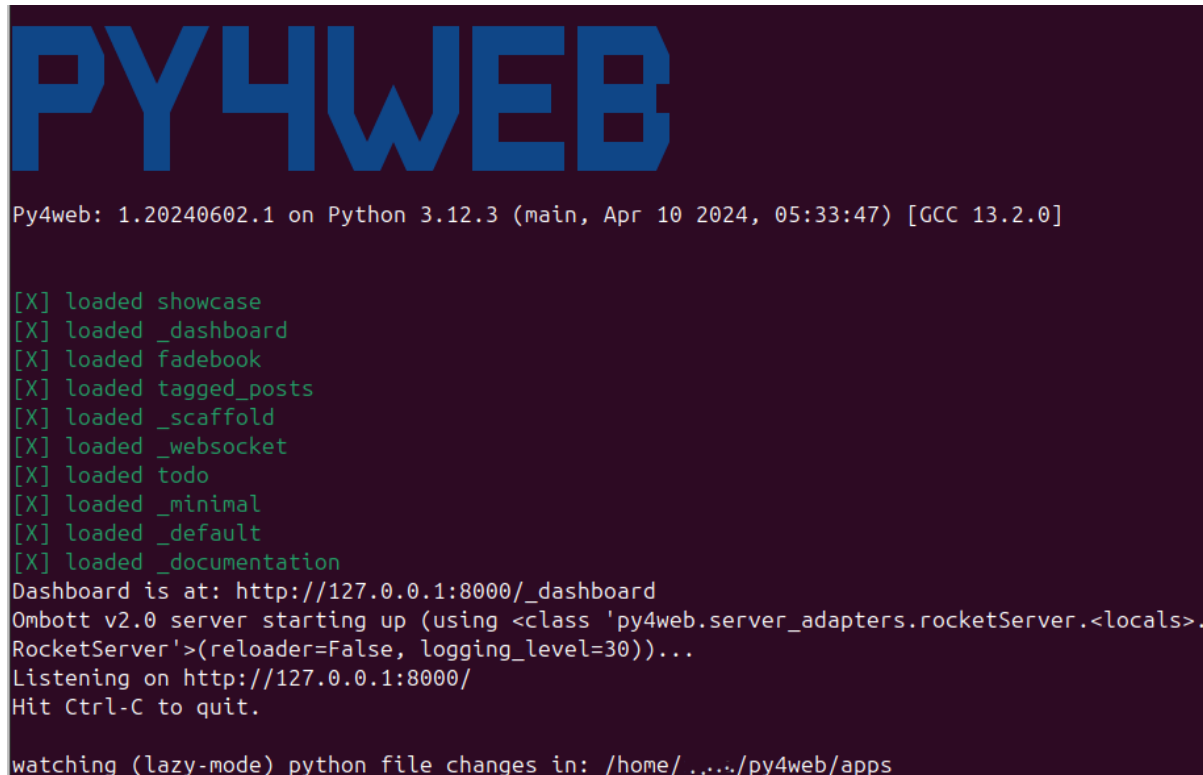
This is the newest way to manage python packages. Install uv as shown here: <https://docs.astral.sh/uv/getting-started/installation/> Then run:

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

More uv command examples are in the provided Makefile

3.5 Primeira corrida

Correndo py4web utilizando qualquer um procedimento anterior deve produzir uma saída como esta:

A terminal window with a dark purple background. At the top, the text 'PY4WEB' is displayed in large, bold, blue capital letters. Below it, the version and environment information is shown: 'Py4web: 1.20240602.1 on Python 3.12.3 (main, Apr 10 2024, 05:33:47) [GCC 13.2.0]'. A list of applications is shown with green status indicators: '[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', and '[X] loaded _documentation'. The dashboard URL is printed: 'Dashboard is at: http://127.0.0.1:8000/_dashboard'. The Ombott server startup message follows: 'Ombott v2.0 server starting up (using <class \'py4web.server_adapters.rocketServer.<locals>.RocketServer\'>(reloader=False, logging_level=30))...'. The listening address is shown: 'Listening on http://127.0.0.1:8000/'. A prompt to hit Ctrl-C to quit is displayed: 'Hit Ctrl-C to quit.'. At the bottom, the file watching mode is active: '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 with 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.

**** Painel **** é um baseado na web IDE. Ele será descrito no próximo capítulo.

**** Padrão **** é um aplicativo que não faz nada diferente de boas-vindas ao usuário.

Note Alguns aplicativos - como o Dashboard **** **** e padrão **** **** - têm um papel especial na py4web e, portanto, seus começos nome real com `` _ `` para evitar conflitos com aplicativos criados por você.

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
```

A fim de py4web stop, você precisa acertar: kbd: *Control-C* na janela onde você executá-lo.

Note Somente o padrão **** **** aplicativo é especial porque se não exige que o "{AppName} /" prefixo no caminho, como todos os outros aplicativos fazer. Em geral, você pode querer ligar simbolicamente `` apps / _default `` ao seu aplicativo padrão.

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 Opções de linha de comando

py4web fornece várias opções de linha de comando que podem ser listados por executá-lo sem qualquer argumento

```
# 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
```

Você pode ter ajuda adicional para uma opção de linha de comando específico, executando-o com o **** - ajuda **** ou **** - h **** argumento.

3.6.1 Opção `` comando call``

```
# 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 Opção `` comando new_app``

```
# 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.
```

Presentemente, dá um erro em instalações binários e de instalação de origem (no local), porque eles perdem o arquivo zip de ativos.

3.6.3 Opção `` comando run``

```
# 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]
```

<code>--watch [off sync lazy]</code>	Watch python changes and reload apps automatically, modes: off, sync, lazy [default: lazy]
<code>--ssl_cert PATH</code>	SSL certificate file for HTTPS
<code>--ssl_key PATH</code>	SSL key file for HTTPS
<code>--errorlog TEXT</code>	Where to send error logs (:stdout :stderr tickets_only {filename}) [default: :stderr]
<code>-L, --logging_level INTEGER</code>	The log level (0 - 50) [default: 30 (=WARNING)]
<code>-D, --debug</code>	Debug switch
<code>-U, --url_prefix TEXT</code>	Prefix to add to all URLs in and out
<code>-m, --mode TEXT</code>	default or development [default: default]
<code>-help, -h, --help</code>	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 unneeded 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 Opção `` comando set_password``

```
# 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.
```

Se o ``-dashboard_mode`` não é ``demo`` ou ``None``, cada vez py4web é iniciado, ele pede uma

senha de uso único para você acessar o painel. Isso é chato. Você pode evitá-lo, armazenando uma senha pdkdf2 hash em um arquivo (por padrão chamado password.txt) com o comando

```
py4web set_password
```

Não vou pedir de novo a menos que o arquivo é excluído. Você também pode usar um nome de arquivo personalizado com

```
py4web set_password my_password_file.txt
```

e depois pedir py4web para reutilização essa senha em tempo de execução com

```
py4web run -p my_password_file.txt apps
```

Finalmente, você pode criar manualmente o mesmo arquivo com:

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

3.6.5 Opção `` comando setup``

```
# 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.
```

Esta opção criar uma nova pasta Aplicativos (ou reinstalá-lo). Se necessário, ele irá pedir a confirmação da criação da nova pasta e, em seguida, para copiar todos os aplicativos py4web padrão da pasta de ativos. Atualmente, não faz nada em instalações binários e de instalação de origem (localmente) - para eles você pode copiar manualmente a pasta de aplicações existentes para o novo.

3.6.6 Opção `` comando shell``

```
# 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.
```

O shell de Py4web é apenas o shell python regular com aplicativos adicionados ao caminho de pesquisa. Note que o shell é para todos os aplicativos, não um único. Você pode então importar os módulos necessários a partir dos aplicativos que você precisa para acessar.

Por exemplo, dentro de uma concha que puder

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

3.6.7 Opção `` comando version``

```
# 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}".

Em seu sistema de arquivos local fazer uma nova pasta de trabalho e cd para ele:

```
mkdir gae
cd gae
```

Copie os arquivos de exemplo de py4web (supondo que você tem a fonte de github)

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

Copiar ou ligar simbolicamente o seu ``apps`` pasta para a pasta gae, ou talvez fazer novos aplicativos pasta que contém um ``__init___.py`` e ligar simbolicamente os aplicativos individuais que você deseja implantar. Você deve ver os seguintes arquivos / pastas:

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

Instale o Google SDK, py4web e configure a pasta de trabalho:

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

(Substitua {seu email} sua conta do Google e-mail e {id projeto} com o ID de projeto obtida de Google).

Agora cada vez que você deseja implantar seus aplicativos, basta fazer:

```
make deploy
```

Você pode querer personalizar o Makefile e app.yaml para atender às suas necessidades. Você não deve precisar editar ``main.py``.

3.7.4 Implantação em 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.

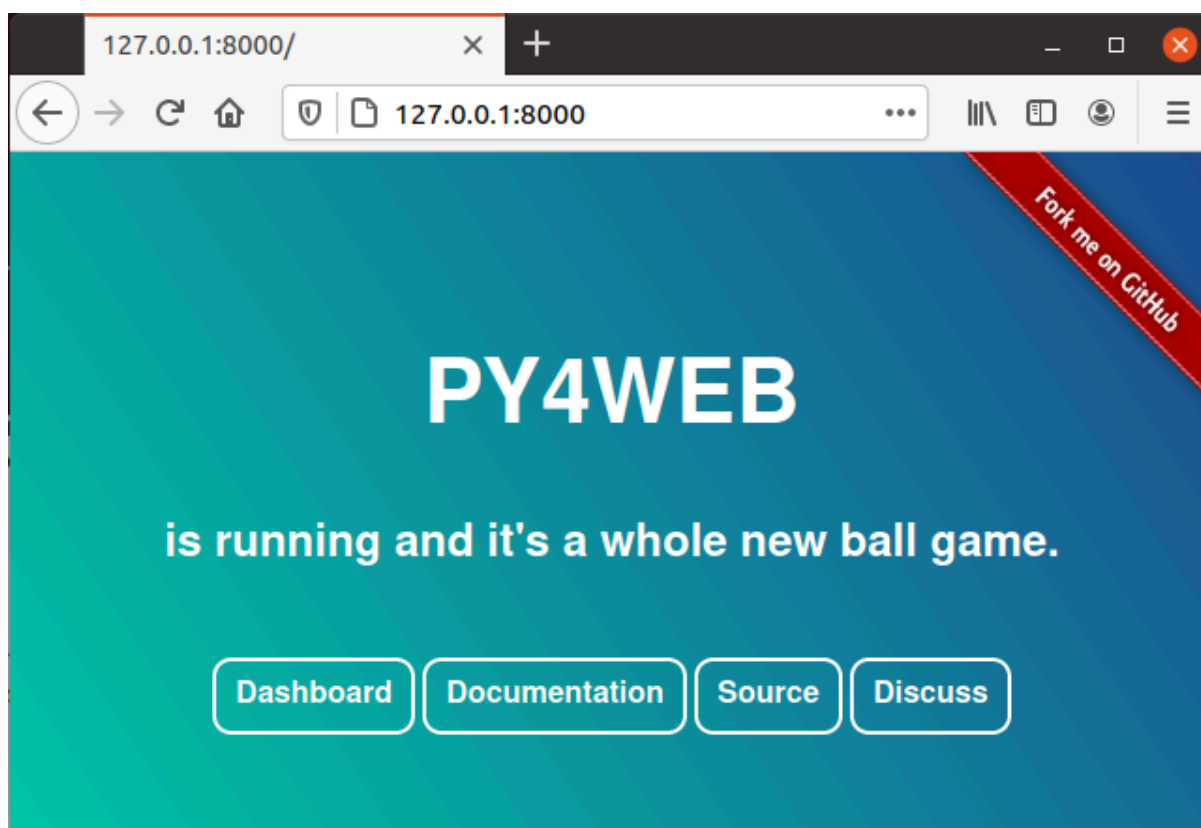
O Dashboard

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 A página Web principal

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:



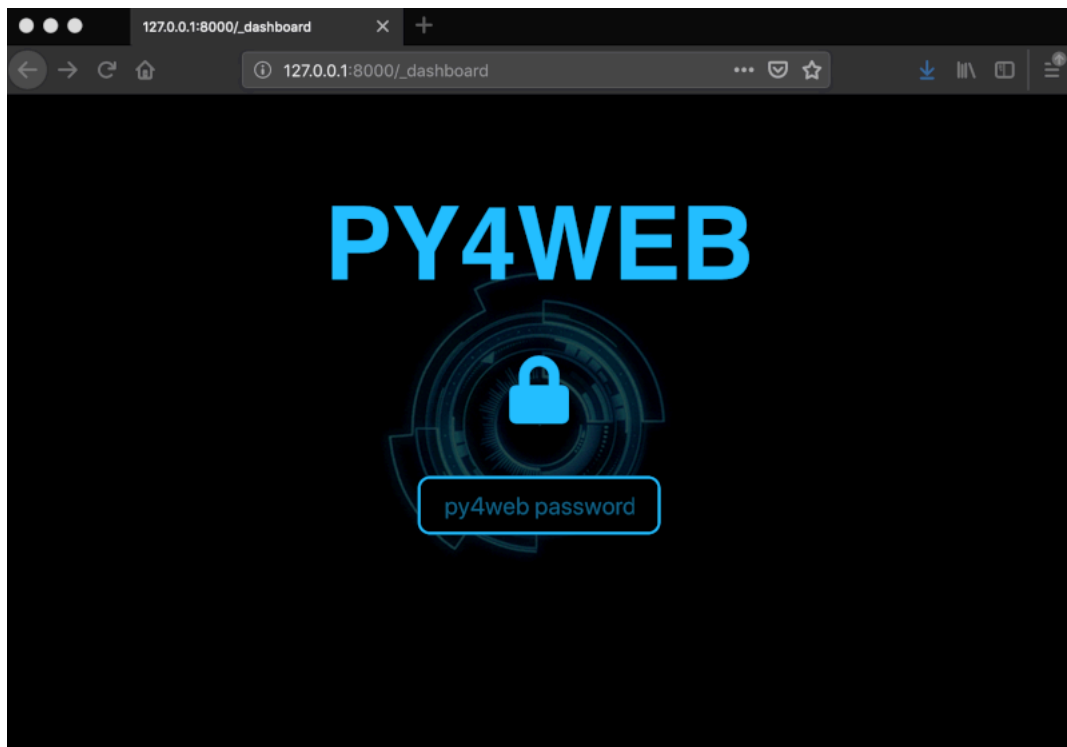
Os botões são:

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

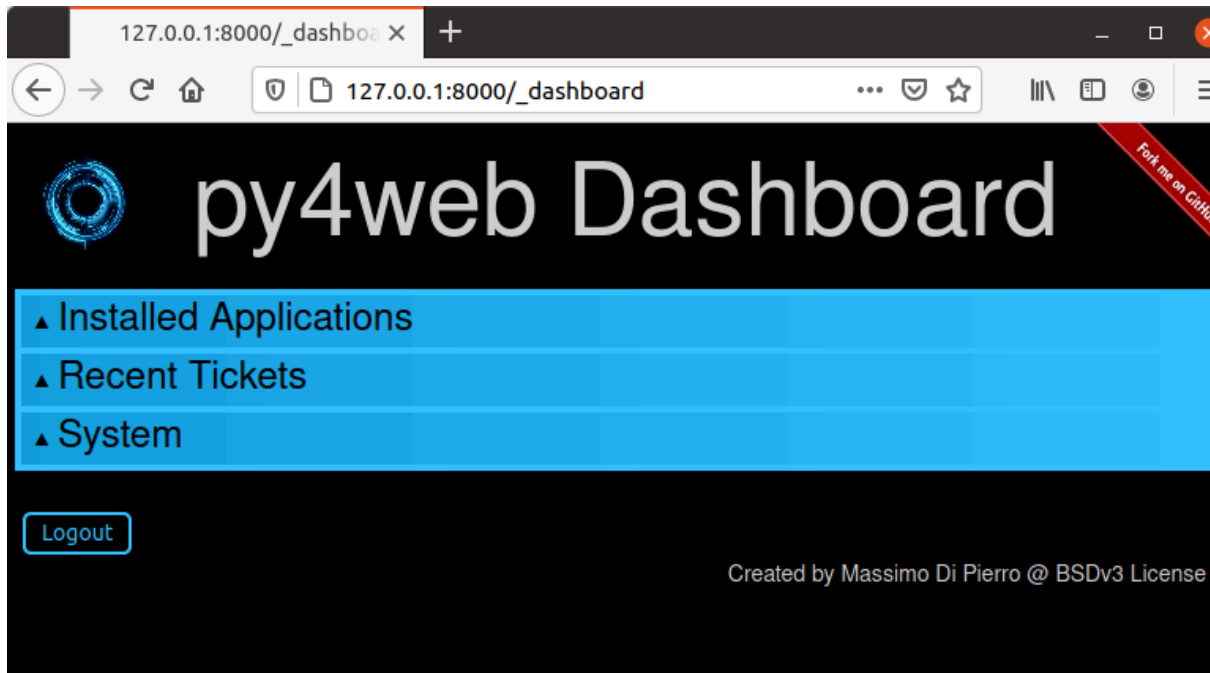
- Documentation (http://127.0.0.1:8000/_documentation?version=1.20201112.1), for browsing the local copy of this Manual.
- Source (<https://github.com/web2py/py4web>), pointing to the GitHub repository.
- Discuss (<https://groups.google.com/forum/#!forum/py4web>), pointing to the Google mail group.

4.2 Sessão no Dashboard

Pressionando o botão do painel irá transmitir-lhe para o login Dashboard. Aqui você deve inserir a senha que você já setup (veja: ref: *option* comando `set_password`). Se você não se lembra da senha, você tem que parar o programa com CTRL-C, configurar um novo e execute o py4web novamente.

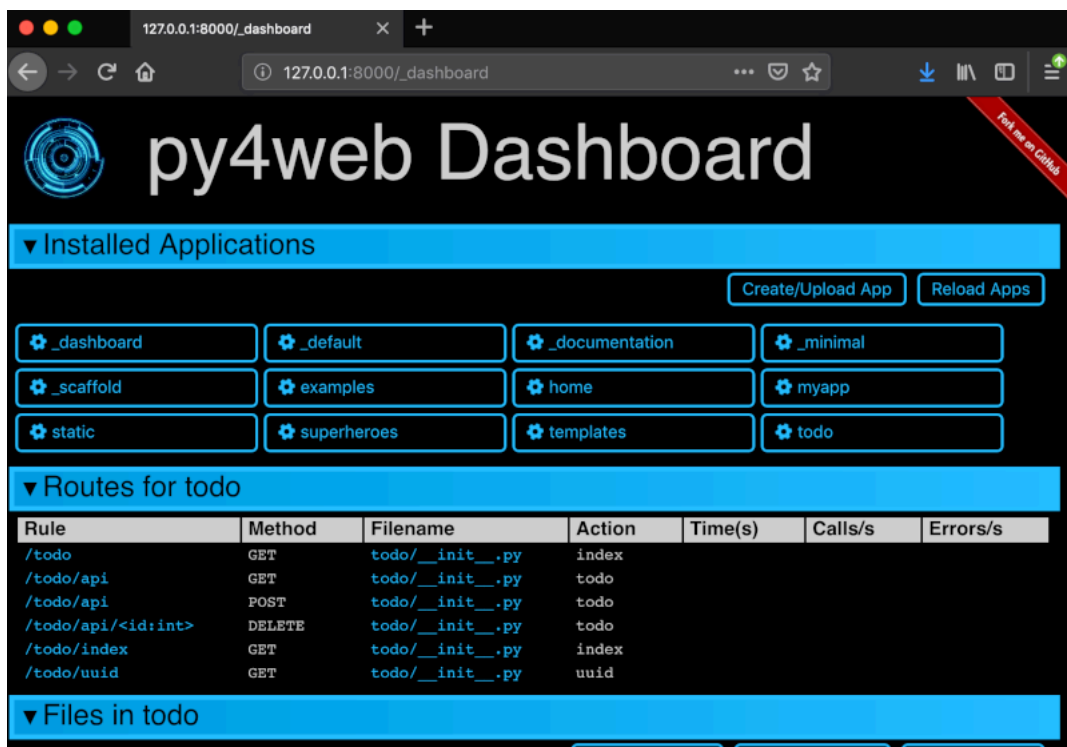


Depois de inserir a senha do painel direito, será exibido com todas as abas comprimido.



Clique no título de um guia para expandir. As guias são dependentes do contexto. Por exemplo, aba aberta “Instalado Aplicativos” e clique em um aplicativo instalado para selecioná-lo.

Isto irá criar novas guias “Rotas”, “Arquivos” e “Modelo” para o aplicativo selecionado.



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.

```

1 import os
2 from py4web import action, request, DAL, Field, Session, Cache, user_in
3
4 # define session and cache objects
5 session = Session(secret='some secret')
6 cache = Cache(size=1000)
7
8 # define database and tables
9 db = DAL('sqlite://storage.db', folder=os.path.join(os.path.dirname(__file__), 'databases'))
10 db.define_table('todo', Field('info'))
11
12 # example index page using session, template and vue.js
13 @action('index') # the function below is exposed as a GET action
14 @action.uses('index.html') # we use the template index.html to render it
15 @action.uses(session) # action needs a session object (read/write cookies)
16 def index():
17     session['counter'] = session.get('counter', 0) + 1
18     session['user'] = {'id': 1} # store a user in session
19     return dict(session=session)
20
21 # example of GET/POST/DELETE RESTful APIs
22
23 @action('api') # a GET API function
24 @action.uses(session) # we load the session
25 @action.requires(user_in(session)) # then check we have a valid user in session
26 @action.uses(db) # all before starting a db connection
27 def todo():
28     return dict(items=db(db.todo).select(orderby=-db.todo.id).as_list())
29

```

O painel expõe o db de todas as aplicações que utilizam RESTAPI pydal. Ele também fornece uma interface web para realizar operações de busca e CRUD.

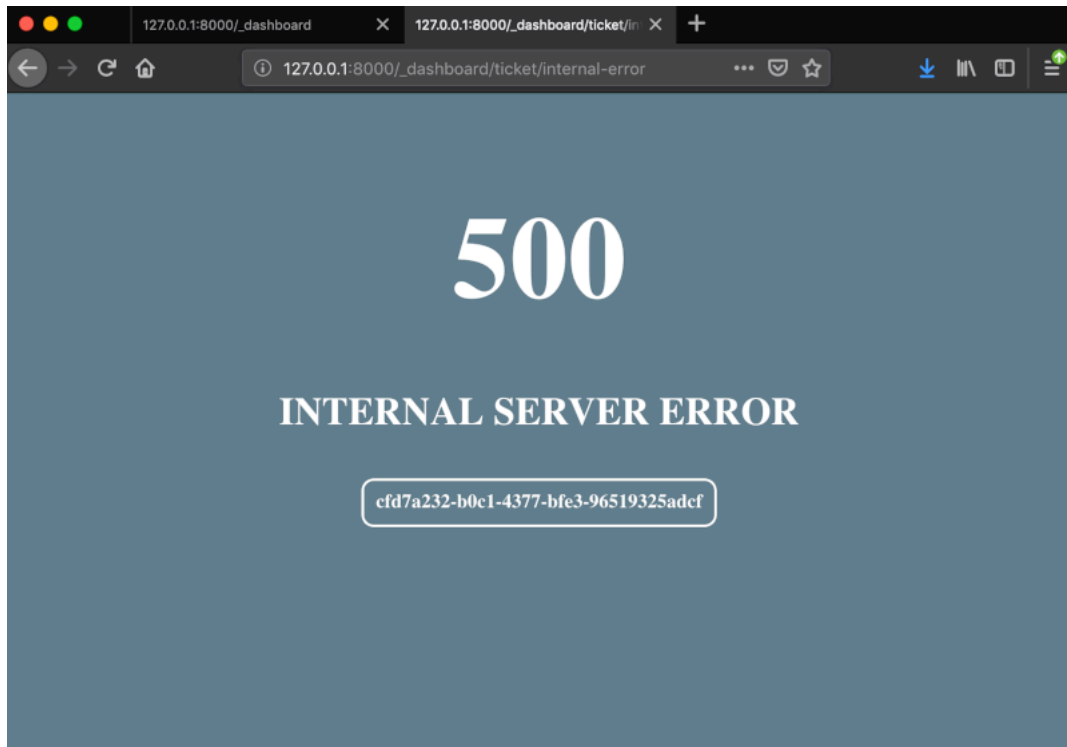
App: superheroes Database: db Table: perso

filter (example id > 1)

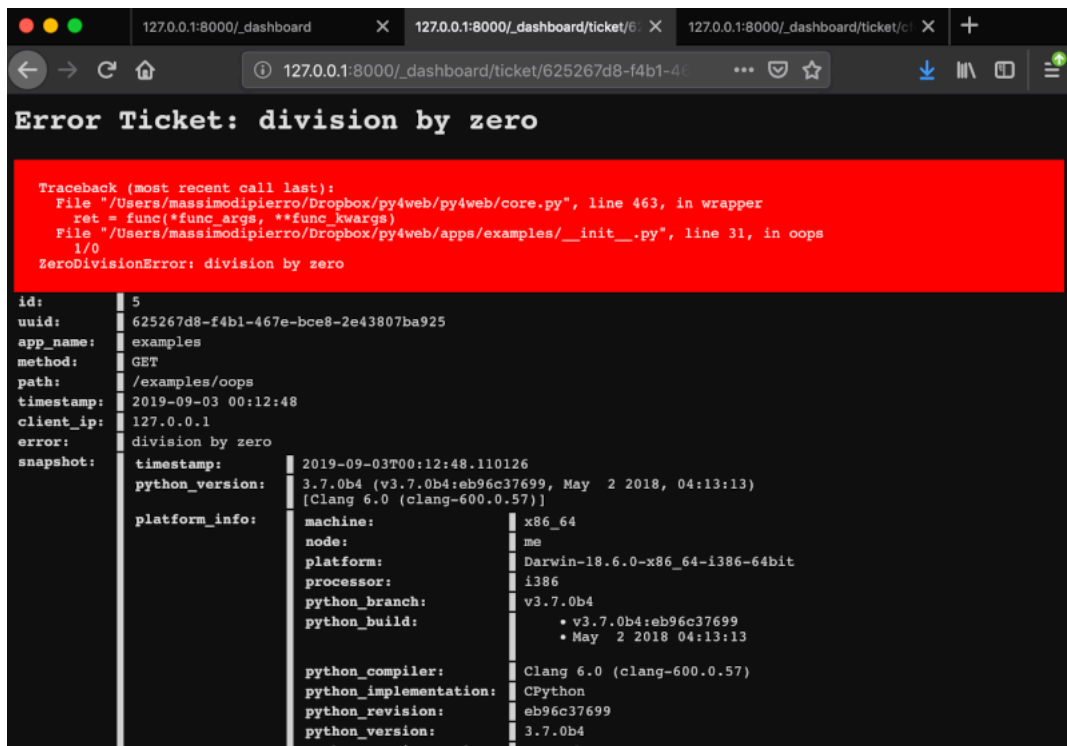
Id	3
Name	Bruce Wayne
Job	CEO
Referenced By	superhero.real_identity

Buttons: Delete, Close

Se um usuário visita um aplicativo e desencadeia um erro, o usuário é emitido um bilhete.



O bilhete é registrado no banco de dados py4web. O painel exibe as edições recentes mais comuns e permite pesquisar bilhetes.



Creating an app

5.1 Do princípio

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 Páginas estáticas

Para expor páginas estáticas você simplesmente precisa para criar um ``subpasta static`` e qualquer arquivo lá será automaticamente publicado:

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

O arquivo recém-criado será acessível em

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

Note que ``static`` é um caminho especial para py4web e arquivos somente sob o ``static`` pasta são servidos.

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 Páginas web dinâmicas

Para criar uma página dinâmica, você deve criar uma função que retorna o conteúdo da página. . Por exemplo editar a ``novaaplicacao / __ __ Init py`` como se segue:

```
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
```

ou

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

(Note que o índice é opcional)

Ao contrário de outras estruturas, nós não importar ou iniciar o servidor web dentro do `` código myapp``. Isso ocorre porque py4web já está em execução, e pode servir vários aplicativos. py4web importa nossas funções de código e expõe decorados com `` @Action () . Note também que prepends py4web `` / myapp (ou seja, o nome do aplicativo) para o caminho url declarado na ação. Isso ocorre porque existem vários aplicativos, e eles podem definir rotas conflitantes. Antecedendo o nome do aplicativo remove a ambiguidade. Mas há uma exceção: se você chamar seu aplicativo `` _default``, ou se você criar um link simbólico do `` _default`` para `` myapp``, então py4web não irá anteceder qualquer prefixo para as rotas definidas dentro do aplicativo .

5.3.1 Em valores de retorno

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

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

Esta página será visível na

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

e retorna um objeto JSON `` { «cores»: [«vermelho», «azul», «verde»]} ``. Observe que escolhemos nomear a função o mesmo que a rota. Isso não é necessário, mas é uma convenção que muitas vezes se seguirão.

Você pode usar qualquer linguagem de modelo para transformar seus dados em uma string. PY4WEB vem com yatl, um capítulo inteiro será dedicado mais tarde e iremos fornecer um exemplo em breve.

5.3.2 Rotas

É possível mapear padrões do URL em argumentos da função. Por exemplo:

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

Esta página será visível na

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

A sintaxe dos padrões é o mesmo que os rotas Garrafa <<https://bottlepy.org/docs/dev/tutorial.html#request-routing>> __. Uma rota WildCard pode ser definida como

- ``<Name>`` ou
- ``<Name: filter>`` ou
- <name:filter:config>

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

- : Resultados int dígitos (assinatura) e converte o valor de número inteiro.
- : Float semelhante a: int mas para números decimais.
- : Path corresponde a todos os personagens, incluindo o caractere de barra de uma forma não-ganancioso, e pode ser usado para combinar mais de um segmento de caminho.
- :re[:exp] allows you to specify a custom regular expression in the config field. The matched value is not modified.

O padrão de harmonização o carácter universal é passado para a função sob a variável especificada ``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
```

Além disso, o decorador acção tem um argumento ``method`` opcional que pode ser um método

HTTP ou uma lista de métodos:

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

Você pode usar vários decoradores para expor a mesma função em várias rotas.

5.3.3 O objeto ``request``

De py4web você pode importar ``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'
```

Esta ação pode ser acessado em:

```
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 Modelos

Para utilizar um yatl modelo que você deve declará-lo. Por exemplo, criar um arquivo ``apps / myapp / templates / paint.html`` que contém:

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

em seguida, modificar a ação de tinta para usar o modelo e padrão para verde.

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

A página irá agora mostrar o nome da cor em um fundo da cor correspondente.

O ingrediente chave aqui é o decorador ``@ action.uses (...)``. Os argumentos de ``action.uses`` são chamados luminárias **** ****. Você pode especificar vários dispositivos elétricos em um decorador ou você pode ter vários decoradores. Chaves são objectos que modificam o comportamento da acção, que podem precisar de ser inicializado por pedido, que podem realizar uma filtragem de entrada e de saída da acção, e que pode depender de cada-outro (eles são semelhantes no seu âmbito à garrafa encaixes, mas eles são declarados por ação, e eles têm uma árvore de dependência que será explicado mais tarde).

O tipo mais simples de acessório é um modelo. Você especifica que simplesmente dando o nome do arquivo a ser usado como modelo. Esse arquivo deve seguir a sintaxe yatl e deve estar localizado no diretório ``templates`` pasta do aplicativo. O objeto retornado pela ação serão processados pelo

modelo e se transformou em uma corda.

Você pode facilmente definir luminárias para outras linguagens de modelo. Isto é descrito mais tarde.

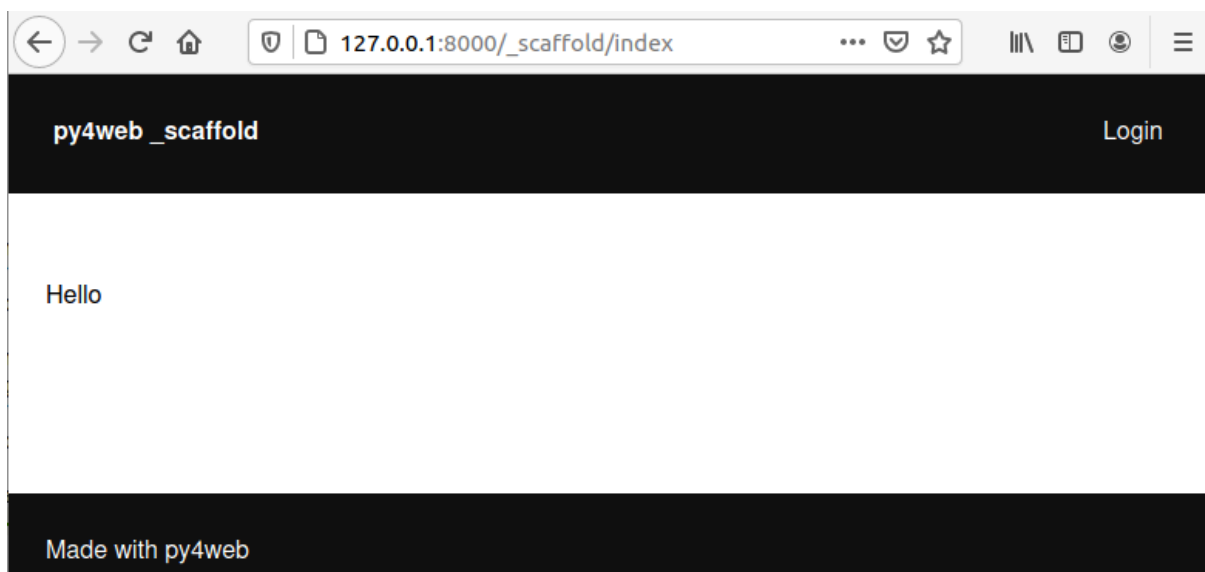
Alguns built-in luminárias são:

- o objeto DAL (que diz py4web para obter uma conexão de banco de dados a partir da piscina a cada pedido, e comprometer-se em caso de sucesso ou reversão em caso de falha)
- o objeto de sessão (que diz py4web para analisar o cookie e recuperar uma sessão a cada pedido, e para salvá-lo, se alterado)
- o objeto Tradutor (que diz py4web para processar o cabeçalho Accept-Language e determinar ótima internacionalização / pluralização regras)
- o objeto Auth (que diz py4web que as necessidades de aplicativos acessar às informações do usuário)

Eles podem depender um do outro. Por exemplo, a sessão pode precisar a DAL (ligação de base de dados), e Auth podem precisamos de ambos. As dependências são tratados automaticamente.

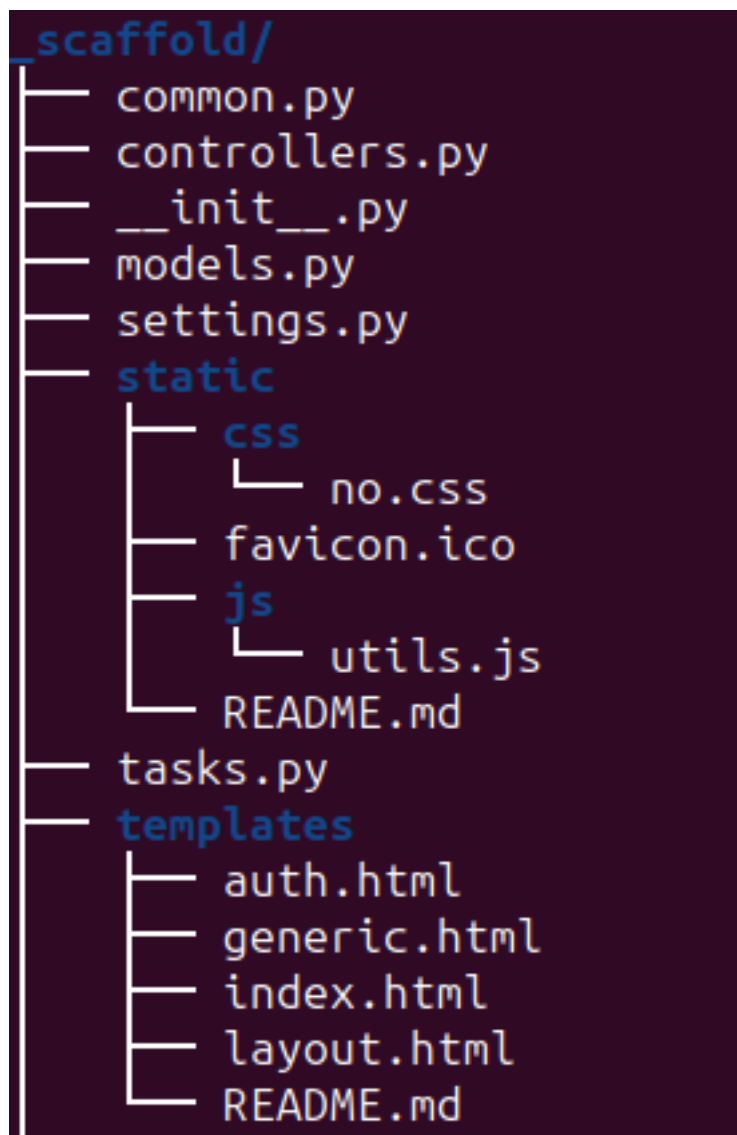
5.4 The `_scaffold` app

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.



Normalmente você vai encontrar o aplicativo andaime sob apps, mas você pode facilmente criar um novo clone de la manualmente ou usando o Dashboard.

Aqui está a estrutura da árvore do `` aplicativo `_scaffold```:



O aplicativo andaime contém um exemplo de uma ação mais complexa:

```

from py4web import action, request, response, abort, redirect, URL
from yat1.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)
  
```

Observe o seguinte:

- 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 yat1.helpers . They work pretty much as in web2py.
- `` Db``, `` session``, `` T``, `` cache``, `` auth`` são Chaves. Eles devem ser definidos em `` common.py``.

- `` @ Action.uses (auth.user) `` indica que esta acção espera um válido logado recuperáveis usuário por `` auth.get_user () . Se isso não for o caso, esta ação redireciona para a página de login (definido também em `` common.py e usando o componente auth.html Vue.js).

Quando você começar a partir de andaime, você pode querer editar `` settings.py``, `` templates``, `` models.py`` e `` controllers.py`` mas provavelmente você não precisa mudar nada no `` common.py``.

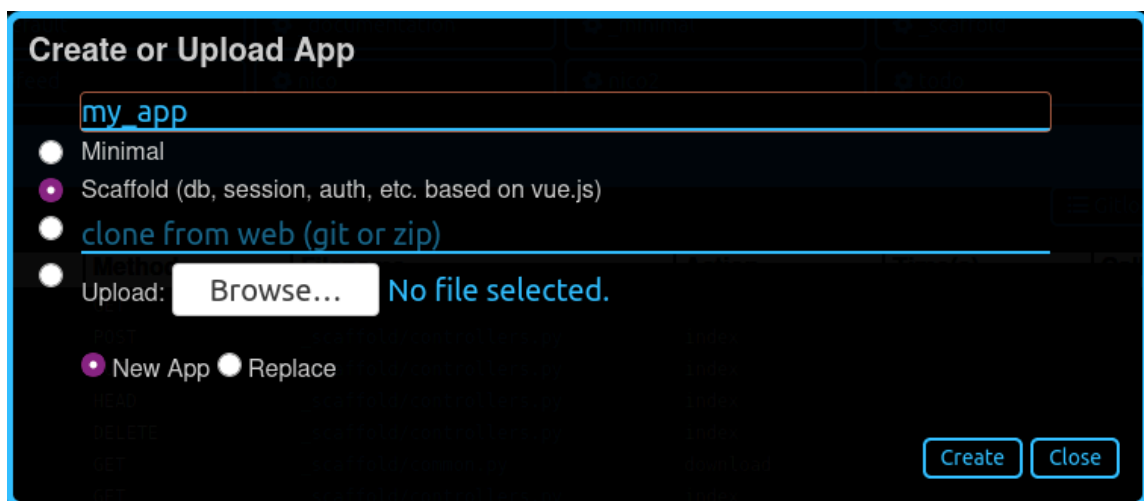
Em seu HTML, você pode usar qualquer biblioteca JS que você quer, porque py4web é agnóstica para a sua escolha de JS e CSS, mas com algumas exceções. O `` auth.html`` que lida com registro / login / etc. usa um componente vue.js. Portanto, se você quiser usar isso, você não deve removê-lo.

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.

Assista SASS arquivos e compilá-los quando editado:

```
from py4web.core import app_watch_handler
```

```
import sass # https://github.com/sass/libsass-python

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

Validar sintaxe javascript quando editado:

```
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 yat1.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

Um fixture é definido como “uma peça de equipamento ou de mobiliário, que é fixa em posição num edifício ou veículo”. No nosso caso, um dispositivo elétrico é algo ligado à ação que processa um pedido HTTP, a fim de produzir uma resposta.

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.

Fixtures PY4WEB são semelhantes aos middleware WSGI e BottlePy plug-in, exceto que eles se aplicam a ações individuais, não para todos eles, e pode dependem uns dos outros.

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 need 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.

```
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.

O objeto template é um dispositivo elétrico. Ele transforma o ``dict ()`` *retornado pela ação em uma string usando o arquivo template index.html*. Em um capítulo posterior iremos fornecer um exemplo de como definir um fixture personalizado para usar uma linguagem de template diferente, por exemplo Jinja2.

Tenha em conta que uma vez que o uso de templates é muito comum e uma vez que, muito provavelmente, cada ação usa um template diferente, nós fornecemos um pouco de açúcar sintático, e as duas linhas a seguir são equivalentes:

```
@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:

```
@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:

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

And then:

```
@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:

```
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

Aqui está um exemplo de uso:

```
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.

Aqui “Translator” é uma classe py4web que se estende “pluralize.Translator” e também implementa a interface de “Fixture”.

Podemos facilmente combinar vários Fixtures. Aqui, como exemplo, podemos tornar a ação com um contador que conta “visitas”.

```
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:

```
@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()
```

Agora crie o seguinte arquivo de tradução ``traduções / 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:

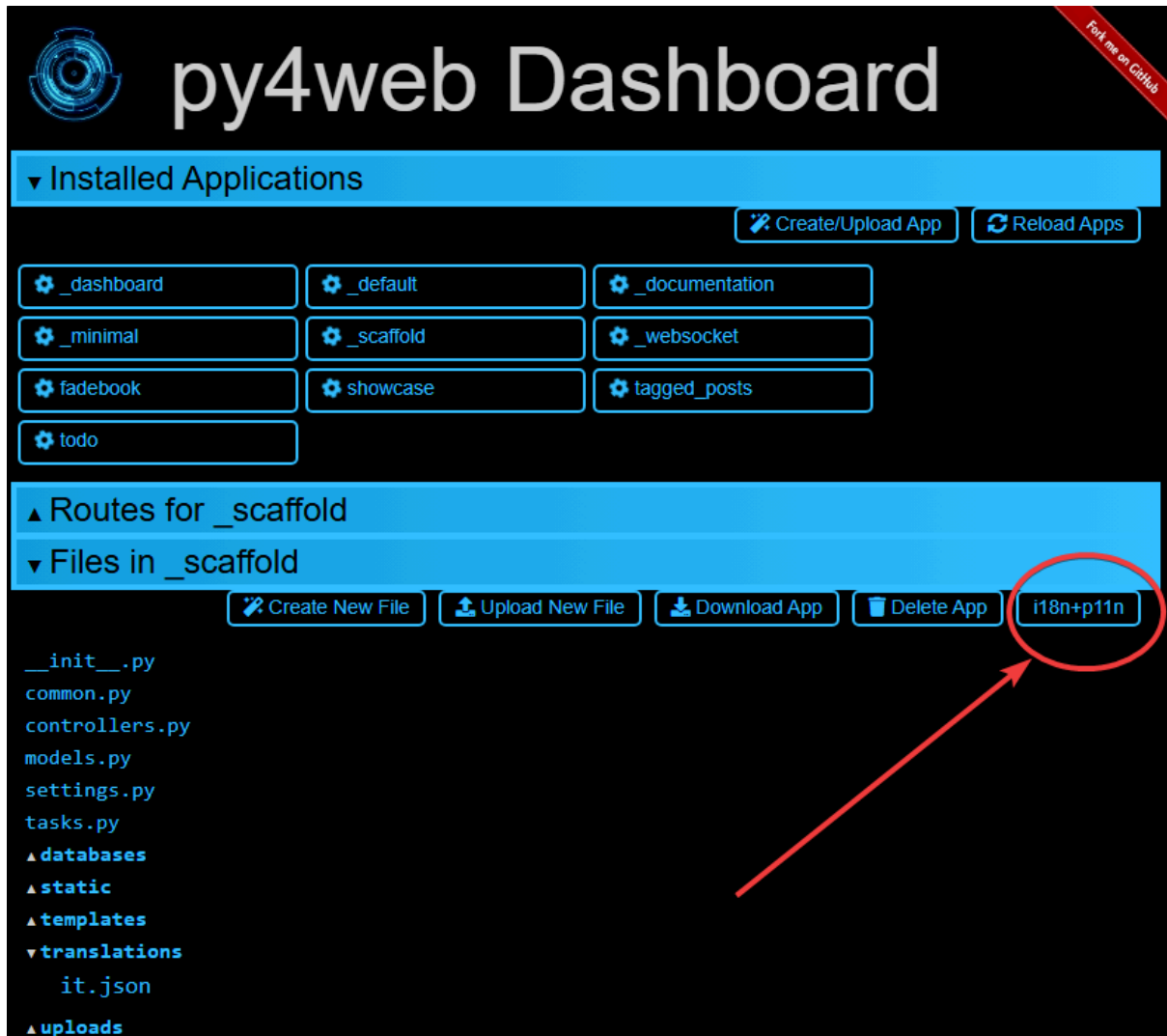
```
This your first time here
You have been here once before
You have been here twice before
You have been here 3 times
You have been here 4 times
You have been here 5 times
You have been here more than 5 times
```

Agora tente criar um arquivo chamado ``traduções / it.json`` que contém:

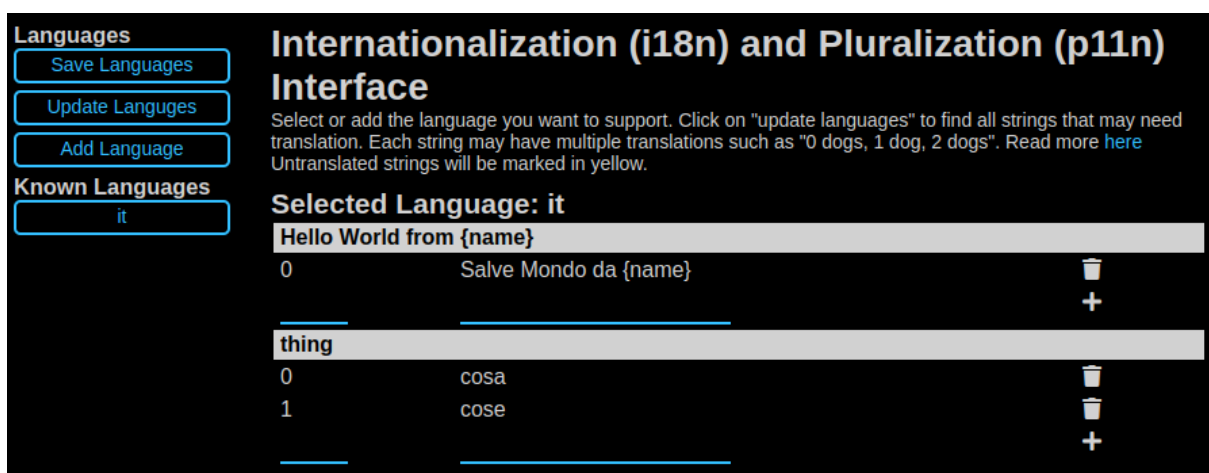
```
{"You have been here {n} times":
  {
    "0": "Non ti ho mai visto prima",
    "1": "Ti ho gia' visto",
    "2": "Ti ho gia' visto 2 volte",
    "3": "Ti ho visto {n} volte",
    "6": "Ti ho visto piu' di 5 volte"
  }
}
```

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 `translations`:



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 O fixture flash

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

O auxiliar o Flash lida com o lado do servidor deles. Aqui está um exemplo:

```
from py4web import Flash

flash = Flash()

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

e no 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.

O cliente também pode definir / adicionar mensagens flash chamando:

```
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

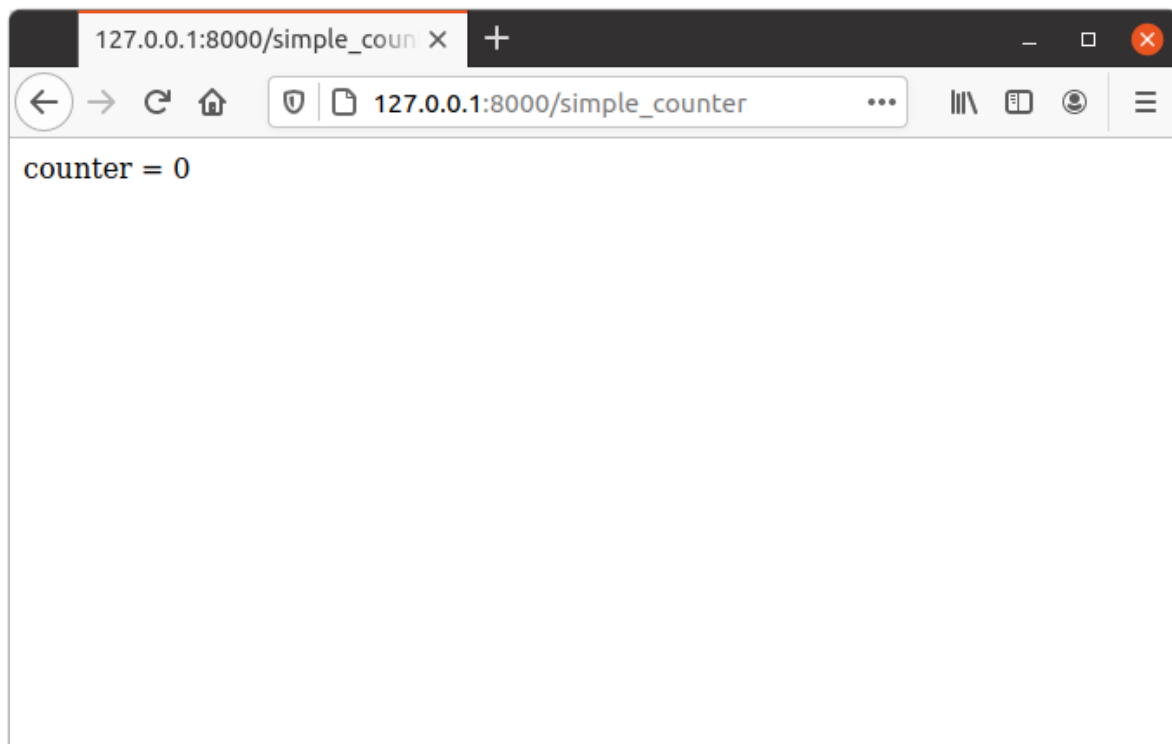
Simply speaking, a session can be defined as a way to preserve information that is desired to persist throughout the user's interaction with the web site or web application. In other words, sessions render the stateless HTTP connection a stateful one.

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

```
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

Por padrão sessões py4web nunca expiram (a menos que contenham informações de login, mas isso é outra história), mesmo se uma expiração pode ser definido. Outros parâmetros podem ser especificados, bem como:

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

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.

```
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.

```
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)
```

Aviso: um objecto de armazenamento deve ter ``GET`` e métodos `set` e do método `set` deve permitir especificar uma expiração. O objecto de ligação redis tem um método `ttl` para especificar a expiração, remendo, portanto, que o macaco `método set` ter a assinatura esperada e funcionalidade.

6.6.4 Server-side session in database

```
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 multiprocessing 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:

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

6.6.5 Server-side session anywhere

Você pode facilmente armazenar sessões em qualquer lugar que você quer. Tudo que você precisa fazer é fornecer ao objeto ``Session`` um objeto ``storage`` com ambos os ``GET`` e ``métodos set``. Por exemplo, imagine que você deseja armazenar sessões no seu sistema de arquivos local:

```
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:

```
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 O fixture DAL

Nós já usou o ``dispositivo elétrico DAL`` no contexto das sessões, mas talvez você queira ter acesso direto ao objeto DAL com a finalidade de acessar o banco de dados, e não apenas sessões.

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:

```
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')
```

O construtor do objeto ``Auth`` define a tabela ``auth_user`` com os seguintes campos: nome de usuário, e-mail, senha, `first_name`, `last_name`, `sso_id` e `action_token` (os dois últimos são principalmente para uso interno).

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:

```
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.

Desde essa verificação é muito comum, py4web fornece um fixture adicional ``auth.user``:

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

This fixture automatically redirects to the `auth/login` page if user is not logged-in, 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

Desde fixtures são compartilhados por várias ações que você não tem permissão para alterar seu estado, porque não seria seguro para threads. Há uma exceção a esta regra. As ações podem alterar alguns atributos de campos de banco de dados:

```
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.

A ``readable``, ``writable``, ``default``, ``update``, e atributos ``require`` de db. {Tabela}. {Campo} `` são objectos especiais de classe `` ThreadSafeVariable`` definido a `` threadsafevariable`` módulo. Esses objetos são muito parecidos com Python rosca objetos locais, mas eles estão em todos os pedidos utilizando o valor fora da ação especificada inicializado-re. Isto significa que as ações podem mudar com segurança os valores desses atributos.

6.12 Fixtures personalizados

Um fixture é um objecto com a seguinte estrutura mínima:

```
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:

```
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:

```
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.HTTPResponse`) 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:

```
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:

```
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:

```
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 `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 e 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 Decoradores de conveniência

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

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

e

```
@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 chosen 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 ou pySqlite2 ou zxJDBC (em Jython)
PostgreSQL	psycopg2 ou zxJDBC (em Jython)
MySQL	pymysql ou MySQLdb
Oráculo	cx_Oracle
MSSQL	pyodbc ou pypyodbc
FireBird	KInterbasDB ou FDB ou pyodbc
DB2	pyodbc
Informix	informixdb
Ingres	ingresdbi
CUBRID	cubridb
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

define py4web as seguintes classes que compõem o 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'))
```

Os métodos mais importantes de uma tabela são:

```Insert``, ``truncate``, ``drop``, e ``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 Usando o 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 Experimentar com o shell py4web

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.

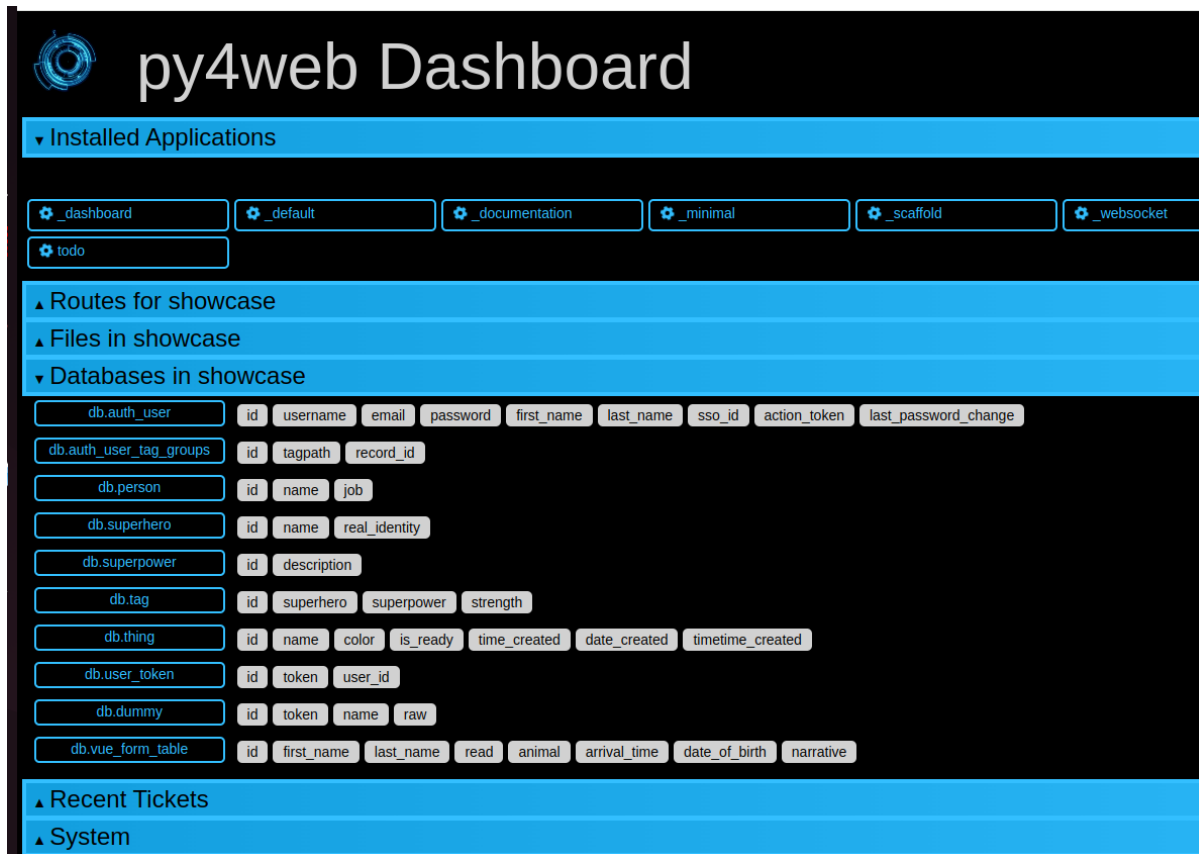
## 7.2 Using the dashboard app with databases

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 showcase 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 Construtor DAL

Uso básico:

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

O banco de dados agora está conectado e a conexão é armazenado na variável global ``db``.

A qualquer momento você pode recuperar a string de conexão.

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

e o nome do banco

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

The connection string is called `_uri` because it is an instance of a uniform resource identifier.

A DAL permite várias ligações com o mesmo banco de dados ou com diferentes bases de dados, mesmo bases de dados de diferentes tipos. Por enquanto, vamos supor a presença de um único banco de dados uma vez que esta é a situação mais comum.

### 7.3.1 Assinatura da DAL

```
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 Strings de conexão (o parâmetro uri)

Uma ligação com o banco de dados é estabelecida através da criação de uma instância do objecto DAL:

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

``Db`` não é uma palavra-chave; é uma variável local que armazena o objeto de conexão ``DAL``. Você é livre para dar-lhe um nome diferente. O construtor de ``DAL`` requer um único argumento, a string de conexão. A sequência de conexão é o único código py4web que depende de um banco de dados específico back-end. Aqui estão alguns exemplos de strings de conexão para tipos específicos de bancos de dados de back-end suportados (em todos os casos, assumimos o banco de dados está sendo executado a partir de localhost na sua porta padrão e é chamado de "teste"):

Database	Connection string
*** SQLite	`` SQLite: // storage.sqlite``
*** MySQL	mysql://username:password@localhost/test?set_encoding=utf8mb4
*** PostgreSQL	postgres://username:password@localhost/test
** MSSQL (legado) **	`` 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
<b>Oráculo</b>	`` 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``

<b>** ** Teradata</b>	<code>teradata://DSN=dsn;UID=user;PWD=pass;DATABASE=test</code>
<b>** ** CUBRID</b>	<code>cubrid://username:password@localhost/test</code>
<b>** ** SAPDB</b>	<code>`` Sapdb: // username: password @ localhost / test``</code>
<b>** ** IMAP</b>	<code>`` Imap: // utilizador: senha @ servidor: port``</code>
<b>** ** MongoDB</b>	<code>mongodb://username:password@localhost/test</code>
<b>** Google / SQL **</b>	<code>`` Google: sql: // projecto: instance / database``</code>
<b>** Google / NoSQL **</b>	<code>`` Google: datastore``</code>
<b>** Google / NoSQL / NDB **</b>	<code>`` Google: armazenamento de dados + ndb``</code>

- 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.

Observe que, por padrão py4web usas utf8 codificação de caracteres para bancos de dados. Se você trabalha com bancos de dados que se comportam de forma diferente existente, você tem que mudá-lo com o parâmetro opcional ``` db_codec``` como

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

Caso contrário, você vai ter bilhetes `UnicodeDecodeError`.

### 7.3.3 O pool de conexões

Um argumento comum do construtor DAL é a ``` pool_size```; o padrão é 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 transac-



tion.If there are no available connections in the pool, a new connection is established.

Quando py4web começa, a piscina é sempre vazio. A piscina cresce até o mínimo entre o valor de ``pool\_size`` e o número máximo de solicitações simultâneas. Isto significa que se ``POOL\_SIZE = 10`` mas o nosso servidor nunca recebe mais de 5 solicitações simultâneas, em seguida, o tamanho real piscina só vai crescer a 5. Se ``POOL\_SIZE = 0`` então o pool de conexão não é usada.

Conexões nas piscinas são compartilhados sequencialmente entre threads, no sentido de que eles podem ser usados por dois tópicos diferentes, mas não simultâneas. Há apenas uma piscina para cada processo py4web.

O parâmetro ``pool\_size`` é ignorado pelo SQLite e Google App Engine. pool de conexão é ignorado para SQLite, uma vez que não daria qualquer benefício.

### 7.3.4 Falhas de conexão (parâmetro tentativas)

Se py4web não consegue se conectar ao banco de dados que espera 1 segundo e por tentativas padrão novamente até 5 vezes antes de declarar um fracasso. No caso do pool de conexão, é possível que uma conexão em pool que permanece aberta, mas sem uso por algum tempo está fechado até o final de banco de dados. Graças à py4web recurso repetição tenta restabelecer essas ligações interrompidas. O número de tentativas é definido através do parâmetro tentativas.

### 7.3.5 Tabelas preguiçosos

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 Aplicativos de modelo-less

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 Bancos de dados replicados

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'])
```

Neste case, as tentativas DAL para conectar-se a primeira e, em case de falha, ele vai tentar o segundo eo terceiro. Isto também pode ser utilizado para distribuir a carga em uma configuração de banco de dados mestre-escravo.

### 7.3.8 Palavras-chave reservadas

``Check\_reserved`` diz o construtor para verificar nomes de tabela e nomes de coluna contra palavras-chave reservada SQL em bancos de dados de back-end-alvo. ``padrões check\_reserved`` a nenhum.

Esta é uma lista de strings que contêm os nomes de adaptador de banco de dados back-end.

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'])
```

A DAL irá analisar as palavras-chave na mesma ordem da lista.

Existem duas opções extras “todos” e “comum”. Se você especificar tudo, ele irá verificar contra todas as palavras-chave SQL conhecidos. Se você especificar comum, ele só irá verificar contra palavras-chave SQL comuns, tais como ``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']
```

Os seguintes backends de banco de dados suportar palavras reservadas verificação.

Database	check_reserved
** ** PostgreSQL	`` Postgres (_nonreserved) ``
** ** MySQL	`` Mysql ``
** ** FireBird	`` Firebird (_nonreserved) ``
** ** MSSQL	`` mssql ``
Oráculo	`` oracle ``

### 7.3.9 Configurações de quoting e case e do banco de dados

Citando de entidades SQL são ativadas por padrão em DAL, isto é:

```
`` Entity_quoting = True ``
```

Desta forma, os identificadores são automaticamente citado em SQL gerado pelo DAL. No SQL palavras-chave de nível e identificadores não cotadas são maiúsculas e minúsculas, quoting assim uma SQL identificador torna maiúsculas de minúsculas.

Note-se que os identificadores não indicada deve sempre ser dobrado para minúsculas pelo motor de back-end acordo com a norma SQL, mas nem todos os motores estão em conformidade com o presente (por exemplo de dobragem PostgreSQL padrão é maiúsculas).

Por DAL padrão ignora case de campo também, para mudar este uso:

```
`` Ignore_field_case = False ``
```

Para ter certeza de usar os mesmos nomes em python e no esquema DB, você deve organizar para ambas as configurações acima. Aqui está um exemplo:

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

### 7.3.10 Fazendo uma conexão segura

Às vezes é necessário (e recomendado) para se conectar ao seu banco de dados usando conexão segura, especialmente se o seu banco de dados não está no mesmo servidor como a sua aplicação. Neste case, você precisa passar parâmetros adicionais para o driver de banco de dados. Você deve consultar a documentação do driver de banco de dados para obter detalhes.

Para PostgreSQL com psycopg2 ele deve ser parecido com isto:

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

onde os parâmetros ``sslrootcert``, ``sslcert`` e ``sslkey`` deve conter o caminho completo para os arquivos. Você deve consultar a documentação do PostgreSQL sobre como configurar o servidor PostgreSQL para aceitar conexões seguras.

### 7.3.11 Outros parâmetros do construtor DAL

#### Local de pasta do banco de dados

`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
```

#### Configurações padrão de migração

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

``Migrar = True`` define o comportamento de migração padrão para todas as tabelas

``Fake\_migrate = False`` define o comportamento fake\_migrate padrão para todas as tabelas

``Migrate\_enabled = True`` se definido como desativa falsas todas as migrações

``Fake\_migrate\_all = False`` Se definido como falso migra Verdadeiros todas as tabelas

### 7.3.12 ``commit`` e 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()
```

Para verificar isso, vamos inserir um novo registro:

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

and roll de volta, ou seja, ignorar todas as operações desde o último commit:

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

Se você agora inserir novamente, o contador voltará a ser definido para 2, desde a inserção anterior foi revertida.

```
>>> 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 Construtor Table

As tabelas são definidos na DAL via `define_table`.

### 7.4.1 assinatura define\_table

A assinatura para o método `define_table` é:

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

Ele aceita um nome de tabela de preenchimento obrigatório e um número opcional de `cases Field` (mesmo nenhum). Você também pode passar um `Table` objeto (ou subclasse) em vez de um `Field` um, este clones e adiciona todos os campos (mas o "id") com a tabela de definição. Outros argumentos de palavra-chave opcionais são: `name`, `redefine`, `common_filter`, `fake_migrate`, `fields`, `format`, `migrate`, `on_define`, `plural`, `polymodel`, `primarykey`, `sequence_name`, `singular`, `table_class`, e `trigger_name`, que são discutidos abaixo.

Por exemplo:

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

Ele define, lojas e retorna um objeto `Table` chamado "pessoa" contendo um campo (coluna) "nome". Este objeto também pode ser acessado via `db.person`, assim você não precisa pegar o valor retornado pelo `define_table`.

### 7.4.2 `id`: Notas sobre a chave primária

Não declare um campo chamado "id", porque um é criado por py4web de qualquer maneira. Cada tabela tem um campo chamado "id" por padrão. É um campo inteiro de auto-incremento (geralmente a partir de 1) utilizados para referência cruzada e para fazer cada registro original, assim que "id" é uma chave primária. (Nota: o contador id a partir de 1 é específico back-end Por exemplo, isto não se aplica ao Google App Engine NoSQL..)

Opcionalmente, você pode definir um campo de `"type = id"` e *py4web usará este campo como campo id auto-incremento. Isso não é recomendado, exceto quando acessar as tabelas de banco de dados legado que têm uma chave primária com um nome diferente. Com alguma limitação, você também pode usar diferentes chaves primárias usando o parâmetro `primarykey`.*

### 7.4.3 `Plural` e `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``

As tabelas podem ser definidas apenas uma vez, mas você pode forçar py4web redefinir uma tabela existente:

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

A redefinição pode provocar uma migração se definição tabela muda.

### 7.4.5 `` Format``: representação da ficha

É opcional, mas recomendado para especificar uma representação formato para registros com o parâmetro `` format``.

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

ou

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

ou mesmo os mais complexos usando uma função:

```
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``: nome real

`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``: Suporte para tabelas legadas

`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``

O nome de uma sequência tabela personalizada (se suportado pelo banco de dados). Pode criar uma sequência (a partir de 1 e incrementando por 1) ou usar isso para tabelas legadas com sequências personalizadas.

Observe que, quando necessário, py4web vai criar sequências automaticamente por padrão.

### 7.4.11 `` Trigger\_name``

Refere-se a `` sequence\_name``. Relevante para alguns backends que não suportam campos numéricos auto-incremento.

### 7.4.12 `` polymodel``

For use with Google App Engine.

### 7.4.13 `` On\_define``

`` On\_define`` é uma chamada de retorno acionado quando um lazy\_table é instanciado, embora ela é chamada de qualquer maneira, se a tabela não é preguiçoso. Isso permite que mudanças dinâmicas para a mesa sem perder as vantagens de instanciação adiada.

Exemplo:

```
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)])
```

Nota Este exemplo mostra como usar `` on\_define`` mas não é realmente necessário. O simples `` valores requires`` poderiam ser adicionados às definições de campo ea mesa ainda seria preguiçoso. No entanto, `` requires`` que tomar um objeto definido como o primeiro argumento, como IS\_IN\_DB, vai fazer uma consulta como `` db.sometable.somefield == some\_value`` que causaria `` sometable`` a ser definido no início . Esta é a situação salvos por `` on\_define``.

### 7.4.14 Adicionando atributos para campos e tabelas

Se você precisa adicionar atributos personalizados aos campos, você pode simplesmente fazer isso: `` 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 Bancos de dados legados e tabelas com chave

py4web pode se conectar a bancos de dados legados sob algumas condições.

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.

Ao acessar uma tabela existente, isto é, uma tabela não criado por py4web no aplicativo atual, sempre definir `` migrar = 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')`).

Finalmente se a tabela de legado usa uma chave primária que não é um campo id auto-incremento é possível usar uma “mesa com chave”, por exemplo:

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

- ``Primarykey`` é uma lista dos nomes de campo que compõem a chave primária.
- Todos os campos PrimaryKey tem um ``NÃO NULL`` definido, mesmo se não especificado.
- Tabelas com chave só podem referenciar outras tabelas com chave.
- Campos de referência devem usar o `reference tablename.fieldname`.
- A ``função update\_record`` não está disponível para filas de mesas com chave.

Atualmente tabelas chaveadas são suportadas apenas para DB2, MSSQL, Ingres e Informix, mas serão adicionados outros engines.

No momento da escrita, não podemos garantir que os ``obras de atributos primarykey`` com cada mesa legado existente e cada backend de banco de dados suportado. Para simplificar, recomendamos, se possível, criando uma visão do banco de dados que tem um campo id auto-incremento.

## 7.5 Construtor Field

Estes são os valores padrão de um construtor de campo:

```
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)
```

onde padrão é um valor especial usado para permitir que o valor Nenhum para um parâmetro.

Nem todos eles são relevantes para todos os campos. ``Length`` é relevante apenas para campos do tipo “string”. ``Uploadfield``, ``authorize``, e ``autodelete`` são relevantes apenas para campos do tipo “Upload”. ``Ondelete`` é relevante apenas para campos do tipo “referência” e “Upload”.

- ``Length`` define o comprimento máximo de uma “string”, “password” ou campo “Upload”. Se ``length`` não for especificado um valor padrão é usado, mas o valor padrão não é garantido para ser compatível. \* Para evitar migrações indesejadas em upgrades, recomendamos que você sempre especificar o comprimento de campos de cordas, senha e upload. \*
- ``Default`` define o valor padrão para o campo. O valor padrão é utilizada quando se realiza uma inserção se um valor não for especificado explicitamente. É também usado para formas construídas a partir da tabela usando ``Form``-preencher previamente. Note, em vez de ser um valor fixo, o padrão em vez disso pode ser uma função (incluindo uma função lambda) que retorna um valor do tipo apropriado para o campo. Nesse case, a função é chamada uma vez para cada registro inserido, mesmo quando vários registros são inseridos em uma única transação.
- ``Required`` conta a DAL que nenhuma inserção deve ser permitido nesta tabela se um valor



para este campo não é especificado explicitamente.

- `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` fornece o campo com um “nome real”, um nome para o campo conhecido para o adaptador de banco de dados; quando o campo é usado, ele é o valor `rname` que é enviado para o banco de dados. O nome `py4web` para o campo é então efetivamente um alias.
- `Ondelete` traduz na “ON DELETE” instrução SQL. Por padrão, ele é definido como “em cascata”. Isso diz ao banco de dados que quando se exclui um registro, ele também deve excluir todos os registros que se referem a ele. Para desativar este recurso, conjunto de `ondelete` a “nenhuma ação” ou “NULL SET”.
- `Notnull = True` se traduz na “NOT NULL” instrução SQL. Ela impede que a banco de dados a partir da inserção de valores nulos para o campo.
- `Unique = True` se traduz na instrução SQL “único” e ele garante que os valores deste campo são exclusivos dentro da tabela. Ela é aplicada no nível de banco de dados.
- `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` permite que você especificar um sistema de arquivos diferente, onde fazer o upload de arquivos, incluindo um armazenamento Amazon S3 ou um armazenamento de SFTP remoto.

Você precisa ter `PyFileSystem` instalado para que isso funcione. `Uploadfs` deve apontar para `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` declara se um campo é gravável em formulários.
- `Readable` declara se um campo é legível em formulários. Se um campo não é nem legível, nem gravável, não será exibido em criar e atualizar formas.
- `Update` contém o valor padrão para este campo quando o registro é atualizado.
- `Compute` é uma função opcional. Se um registro é inserido ou atualizado, a função de



computação será executado eo campo será preenchido com o resultado da função. O registro é passado para a função de computação como um ``dict``, eo dict não incluirá o valor atual de que, ou qualquer outro campo de computação.

- ``Authorize`` pode ser usado para exigir o controle de acesso no campo correspondente, para apenas os campos "Upload". Ele será discutido mais em detalhe no contexto de autenticação e autorização.
- `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 (comprimento) `` comprimento padrão é 512
``text``	`` IS_LENGTH (comprimento) `` comprimento padrão é 32.768
``blob``	`` Comprimento padrão None `` é 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 (comprimento) `` comprimento padrão é 512
``Upload``	`` Comprimento padrão é 512 None ``
``Referência <table> ``	`` IS_IN_DB (db, table.field, formato) ``

<code>`` Lista: string``</code>	<code>`` None``</code>
<code>`` Lista: integer``</code>	<code>`` None``</code>
<code>`` Lista: referência &lt;table&gt; ``</code>	<code>IS_IN_DB(db, table._id, format, multiple=True)</code>
<code>`` json``</code>	<code>`` IS_EMPTY_OR (IS_JSON ()) `` comprimento padrão é 512</code>
<code>`` bigint``</code>	<code>`` IS_INT_IN_RANGE (** -2 63, 2 ** 63) ``</code>
<code>`` Grande-id``</code>	<code>`` None``</code>
<code>`` Grande-reference``</code>	<code>`` None``</code>

Decimal requer e devolve valores como ``` objectos Decimal```, como definidos na Python ``` decimal``` módulo. SQLite não lidar com o ``` decimal``` tipo assim internamente que tratá-lo como um ``` double```. O (n, m) são o número de dígitos no total e o número de dígitos após o ponto decimal, respectivamente.

A ``` grande-id``` e ``` grande-reference``` são suportados apenas por alguns dos mecanismos de bases de dados e são experimentais. Eles não são normalmente usados como tipos de campo a menos de tabelas legadas, no entanto, o construtor DAL tem um argumento ``` bigint_id``` que, quando definido para ``` True``` faz com que os ``` campos id``` e ``` campos reference``` ``` grande-id``` e ``` grande-reference``` respectivamente.

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 modificação da tabela e campo em tempo de execução

A maioria dos atributos de campos e tabelas podem ser modificados depois que eles são definidos:

```
>>> 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'
```

aviso de que os atributos de tabelas são geralmente precedido por um sublinhado para evitar conflitos com possíveis nomes de campo.

Você pode listar as tabelas que foram definidos para uma determinada conexão com o banco:

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

Você pode consultar para o tipo de uma tabela:

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

Você pode acessar uma tabela utilizando diferentes sintaxes:

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

Você também pode listar os campos que foram definidos para uma determinada tabela:

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

Da mesma forma você pode acessar campos de seu nome de várias maneiras equivalentes:

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

Dado um campo, você pode acessar os atributos definidos em sua definição:

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

incluindo a sua tabela pai, tablename, e ligação parent:

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

Um campo também tem métodos. Alguns deles são utilizados para consultas de construção e vamos vê-los mais tarde. Um método especial do objeto de campo é ``validate`` e chama os validadores para o campo.

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

que retorna um tuplo ``(valor, erro)``. ``Error é None`` se a entrada passa a validação.

## 7.5.4 Mais sobre envios

Considere o seguinte modelo:

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

No case de um campo de “carregamento”, o valor padrão pode, opcionalmente, ser definida como um caminho (um caminho absoluto ou um caminho relativo para a pasta aplicativo atual), o valor padrão é então atribuído a cada novo registro que não especifica um imagem.

Observe que desta forma vários registros podem acabar com referência ao mesmo arquivo de imagem padrão e isso poderia ser um problema em um campo ter ``autodelete`` habilitado. Quando você não quer permitir duplicatas para o campo de imagem (ou seja, vários registros referenciando o mesmo arquivo), mas ainda quer definir um valor padrão para o “carregamento”, então você precisa de uma forma de copiar o arquivo padrão para cada novo registro que faz não especificar uma imagem. Isto pode ser obtido usando um arquivo-como objeto referenciando o arquivo padrão como o argumento ``default`` ao campo, ou mesmo com:

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

Normalmente uma inserção é feita automaticamente através de um ``Form`` mas ocasionalmente você já tem o arquivo no sistema de arquivos e quer enviá-lo por meio de programação. Isso pode ser

feito da seguinte maneira:

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

Também é possível inserir um arquivo de uma forma mais simples e tem a chamada de método de inserção ``store`` automaticamente:

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

Neste case, o nome do ficheiro é obtido a partir do objecto corrente, se disponível.

O método `store` do objeto campo carregamento leva um fluxo de arquivo e um nome de arquivo. Ele usa o nome do arquivo para determinar a extensão (tipo) do arquivo, cria um novo nome temporário para o arquivo (de acordo com mecanismo de upload py4web) e carrega o conteúdo do arquivo neste novo arquivo temporário (sob os envios de pasta salvo indicação em contrário). Ele retorna o novo nome temp, que é então armazenada no campo ``image`` da tabela ``db.myfile``.

Note, se o arquivo deve ser armazenado em um campo blob associado ao invés do sistema de arquivos, o método `store` não irá inserir o arquivo no campo blob (porque `store` é chamado antes da inserção), portanto, o arquivo deve ser explicitamente inserido no campo blob:

```
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())
```

O método `retrieve` faz o oposto do `store`.

Quando os arquivos enviados são armazenados no sistema de arquivos (como no case de um ``Field("imagem" simples, "upload")``) o código:

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

recupera o nome do arquivo original (filename) como visto pelo usuário em tempo de upload e o nome do arquivo armazenado (fullname, com caminho relativo para a pasta da aplicação). Embora, em geral, a chamada:

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

recupera o nome original do arquivo (filename) e um arquivo-como objeto pronto para dados de arquivo de acesso carregado (stream).

Observe que o fluxo retornado por ``retrieve`` é um objeto de arquivo real no case de que os arquivos enviados são armazenados no sistema de arquivos. Nesse case, lembre-se de fechar o arquivo quando você é feito, chamando ``stream.close()``.

Aqui está um exemplo de uso seguro do ``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 Migrações

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.

Referimo-nos a esse comportamento como uma “migração”. py4web registra todas as migrações e migração tentativas no arquivo “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')
```

O valor de migrar é o nome do arquivo onde as informações de migração interna para esta tabela. Esses arquivos são muito importantes e nunca deve ser removido enquanto existirem as tabelas correspondentes. Nos casos em que uma tabela foi descartada e o arquivo correspondente ainda existem, ele pode ser removido manualmente. Por padrão, `migrate` é definida como `True`. Este causa py4web para gerar o nome do arquivo a partir de um hash da string de conexão. Se `migrate` é definida como falso, a migração não é realizada, e py4web assume que a tabela existe no armazenamento de dados e que contém (pelo menos) os campos listados no “`define_table`”.

Não pode haver duas tabelas no mesmo aplicativo com o mesmo nome migrar.

A classe DAL também leva um argumento “migrar”, que determina o valor padrão de migrar para chamadas para “`define_table`”. Por exemplo,

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

irá definir o valor padrão de migrar para Falso quando “`db.define_table`” é chamado sem um argumento migrar.

**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`.

As migrações podem ser desativado para todas as tabelas de uma só vez:

```
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 Fixação migrações quebrados

Há dois problemas comuns com as migrações e existem formas de recuperar a partir deles.

Um problema é específico com SQLite. SQLite não impor tipos de coluna e não pode soltar colunas. Isto significa que se você tiver uma coluna do tipo string e você removê-lo, não é realmente removido. Se você adicionar a coluna novamente com um tipo diferente (por exemplo, data e hora) você acaba com uma coluna de data e hora que contém strings (lixo para fins práticos). não py4web não reclamar sobre isso, porque ele não sabe o que está no banco de dados, até que ele tenta recuperar registros e

falha.

Se py4web retorna um erro em alguma função de análise ao selecionar registros, muito provavelmente isso é devido a dados corrompidos em uma coluna por causa da questão acima.

A solução consiste em actualizar todos os registos da tabela e a actualização dos valores na coluna em questão com Nenhum.

O outro problema é mais genérico, mas típico com MySQL. O MySQL não permitem mais de um ALTER TABLE em uma transação. Isto significa que py4web deve quebrar transações complexas em partes menores (um ALTER TABLE na época) e cometem uma peça no momento. Por isso, é possível que parte de uma transação complexa fica comprometida e uma parte falhar, deixando py4web em um estado corrompido. Por que parte de uma transação falhar? Uma vez que, por exemplo, envolve a alteração de uma tabela de conversão e uma coluna de strings dentro de uma coluna de data e hora, tentativas py4web para converter os dados, mas os dados não podem ser convertidos. O que acontece com py4web? Ele fica confuso sobre o que exatamente é a estrutura da tabela realmente armazenados no banco de dados.

A solução consiste em permitir migrações falsos:

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

Isto irá reconstruir metadados py4web sobre a tabela de acordo com a definição da tabela. Tente várias definições de tabela para ver qual delas funciona (aquele antes da migração falhou ea uma após a migração falhou). Uma vez que remove sucesso do ``fake\_migrate = True`` parâmetro.

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

Migração problemas também pode ser fixada para todas as tabelas de uma só vez:

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

Isso também falhará se o modelo descreve tabelas que não existem no banco de dados, mas pode ajudar a estreitar o problema.

## 7.6.2 Migração resumo controle

A lógica dos vários argumentos de migração estão resumidos neste pseudo-código:

```
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``

Dada uma tabela, você pode inserir registros

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

Inserir retorna o valor único "id" de cada registro inserido.

Você pode truncar a tabela, ou seja, excluir todos os registros e reinicie o contador do id.

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

Agora, se você inserir um registro novo, o contador recomeça a 1 (isto é específico back-end e não se aplica ao Google NoSQL):

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

Observe que você pode passar um parâmetro para ``truncate``, por exemplo, você pode dizer SQLite para reiniciar o contador id.

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

O argumento é em SQL puro e, portanto, específico do motor.

py4web também fornece um método `bulk_insert`

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

É preciso uma lista de dicionários de campos a serem inseridas e executa várias inserções ao mesmo tempo. Ele retorna a lista de valores "id" dos registros inseridos. Nos bancos de dados relacionais suportadas não há nenhuma vantagem em usar esta função ao invés de looping e realizando inserções individuais, mas no Google App Engine NoSQL, há uma grande vantagem de velocidade.

## 7.7.2 ``Query``, ``Set``, ``Rows``

Vamos considerar novamente a tabela definida (e caiu) anteriormente e inserir três registros:

```
>>> 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
```

Você pode armazenar a tabela em uma variável. Por exemplo, com variável ``Person``, você poderia fazer:

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

Você também pode armazenar um campo em uma variável como ``name``. Por exemplo, você também pode fazer:

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

Você pode até criar uma consulta (usando operadores como ==, =, <,>, <=,> =, como, pertence!) E armazenar a consulta em uma variável ``q`` tal como em:

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

Quando você chamar ``db`` com uma consulta, você define um conjunto de registros. Você pode armazená-lo em uma variável ``s`` e escreve:

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

Observe que nenhuma consulta de banco de dados foi realizada até agora. DAL + consulta simplesmente definir um conjunto de registros neste db que correspondem a consulta. py4web determina a partir da consulta que tabela (ou tabelas) estão envolvidos e, de fato, não há necessidade de especificar isso.

## 7.7.3 ``Update\_or\_insert``

Algumas vezes você precisa executar uma inserção somente se não há nenhum registro com os

mesmos valores como aqueles que estão sendo inseridos. Isso pode ser feito com

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

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

O registro será inserido somente se não houver nenhum outro usuário chamado John nascido em Chicago.

Você pode especificar quais valores usar como uma chave para determinar se existe o registro. Por exemplo:

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

e se houver John sua terra natal será atualizado então um novo registro será criado.

Os critérios de selecção no exemplo acima é um único campo. Ele também pode ser uma consulta, tais como

```
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``

A função

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

funciona muito bem como

```
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.

Similarmente

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

funciona muito da mesma forma como

```
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``

Finalmente, você pode soltar tabelas e todos os dados serão perdidos:

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



## 7.7.6 QueryBuilder

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

```
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 brackets to group. It throws a QueryParseError 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 explicitly 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 Marcação de registros

Etiquetas permite adicionar ou encontrar propriedades anexadas aos registros em seu banco de dados.

```
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``

A DAL permite emitir explicitamente instruções SQL.

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

Neste case, os valores de retorno não são analisados ou transformados pela DAL, eo formato depende do driver de banco de dados específico. Este uso com selecciona normalmente não é necessário, mas é mais comum com índices.

``Executesql`` leva cinco argumentos opcionais: ``placeholders``, ``as\_dict``, ``fields``, ``colnames``, e ``as\_ordered\_dict``.

``Placeholders`` é uma sequência opcional de valores a serem substituídos ou, se suportado pelo driver DB, um dicionário com chaves correspondentes chamado espaços reservados no seu 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`` é muito bonito como ``as\_dict`` mas os antigos garante que a ordem dos resultando campos (teclas OrderedDict) refletem a ordem em que eles são retornados de DB motorista:

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

O argumento ``fields`` é uma lista de objetos DAL de campo que correspondem aos campos retornados da DB. Os objectos de campo devem ser parte de um ou mais objectos Tabela definido no objecto DAL. A ``lista fields`` pode incluir um ou mais DAL Tabela objetos em adição a ou em vez de incluir Campo objetos, ou pode ser apenas uma única tabela (não de uma lista). Nesse case, os objectos de campo vai ser extraído da tabela (s).

Em vez de especificar o argumento ``fields``, o argumento ``colnames`` pode ser especificado como uma lista de nomes de campos em formato tabela.campo. Novamente, estes devem representar tabelas e campos definidos no objeto DAL.

Também é possível especificar ``fields`` eo associado ``colnames``. Nesse case, ``fields`` pode também incluir objetos Expressão DAL, além de objetos de campo. Para objetos de campo em "campos", o associado ``colnames`` ainda deve estar no formato tabela.campo. Para Expression objetos em ``fields``, o associado ``colnames`` podem ser quaisquer rótulos arbitrários.

Observe, a Tabela DAL objectos referidos por ``fields`` ou ``colnames`` pode ser fictícios mesas e não têm para representar todas as tabelas reais no banco de dados. Além disso, nota que o ``fields`` e ``colnames`` devem estar na mesma ordem que os campos nos resultados cursor retornado do DB.

### 7.8.2 ``\_lastsql``

Se SQL foi executado manualmente usando ExecuteSQL ou foi SQL gerado pelo DAL, você sempre pode encontrar o código SQL em ``db.\_lastsql``. Isso é útil para fins de depuração:

```
>>> 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 Temporização de consultas

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 Índices

Atualmente, a API DAL não fornece um comando para criar índices em tabelas, mas isso pode ser feito usando o comando `executesql`. Isso ocorre porque a existência de índices pode fazer migrações complexa, e é melhor para lidar com eles de forma explícita. Os índices podem ser necessários para esses campos que são usados em consultas recorrentes.

Aqui está um exemplo de como:

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

Outros dialetos banco de dados têm sintaxes muito semelhantes, mas pode não suportar o opcional `“SE NÃO EXISTE”` directiva.

### 7.8.5 Generating raw SQL

Às vezes você precisa para gerar o SQL, mas não executá-lo. Isso é fácil de fazer com py4web uma vez que cada comando que executa banco de dados IO tem um comando equivalente que não, e simplesmente retorna o SQL que teriam sido executados. Estes comandos têm os mesmos nomes e sintaxe como os funcionais, mas eles começam com um sublinhado:

Aqui é `__insert__`

```
>>> print(db.person.__insert(name='Alex'))
INSERT INTO "person" ("name") VALUES ('Alex');
```

Aqui é `__count__`

```
>>> print(db(db.person.name == 'Alex').__count())
SELECT COUNT(*) FROM "person" WHERE ("person"."name" = 'Alex');
```

Aqui é `__select__`

```
>>> print(db(db.person.name == 'Alex').__select())
SELECT "person"."id", "person"."name" FROM "person" WHERE ("person"."name" = 'Alex');
```

Aqui é `__delete__`

```
>>> print(db(db.person.name == 'Alex').__delete())
DELETE FROM "person" WHERE ("person"."name" = 'Alex');
```

E, finalmente, aqui é `__update__`

```
>>> print(db(db.person.name == 'Alex').__update(name='Susan'))
UPDATE "person" SET "name"='Susan' WHERE ("person"."name" = 'Alex');
```

Além disso, você sempre pode usar `db._lastsql` para retornar o código SQL mais recente, se foi executada manualmente usando `ExecuteSQL` ou foi SQL gerado pelo DAL.

## 7.9 `` Comando SELECT``

Dado um conjunto, `` s``, você pode buscar os registros com o comando `` 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.

O objecto `Fileiras` permite loop sobre o resultado do seleccionar e imprimindo os valores dos campos seleccionados para cada linha:

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

Você pode fazer todas as etapas em uma declaração:

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

O comando `select` pode tomar argumentos. Todos os argumentos sem nome são interpretados como os nomes dos campos que você deseja buscar. Por exemplo, você pode ser explícita no campo “id” e no campo “nome” buscar:

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

O atributo `ALL` permite especificar todos os campos:

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

Observe que não há nenhuma strings de consulta passada para `db`. `py4web` entende que se você quiser todos os campos da pessoa mesa sem informações adicionais, então você quer todos os registros da pessoa mesa.

Uma sintaxe alternativa equivalente é o seguinte:

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

e `py4web` entende que se você perguntar para todos os registros da pessoa mesa sem informações adicionais, então você quer todos os campos da tabela pessoa.

Dada uma linha

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

you can extract its values using various 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 allow, instead of this, the less compact notation:

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

Sim, isso é incomum e raramente necessário.

Row objects also have two important methods:

```
row.delete_record()
```

and

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

### 7.9.1 Usando um seletor para uso de memória inferior à base de iterador

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

The traditional use of selection is:

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

but for a large number of lines, using an alternative to the base iterator has the use of memory dramatically inferior:

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

Tests show that this is about 10% faster, even on machines with a lot of RAM.

### 7.9.2 Renderizando Rows com represent

You can want to rewrite lines returned by selection to take advantage of information about formatting contained in the representation of the fields.

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

If you don't specify an index, you have a generator to iterate over all lines:

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

Também pode ser aplicada a fatias:

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

Se você só quer transformar campos selecionados através do seu atributo “representar”, você pode incluí-los no argumento “campos”:

```
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 Atalhos

A DAL suporta vários atalhos-simplificando código. Em particular:

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

retorna o registro com o dado `` id`` se ele existir. Se o `` id`` não existe, ele retorna `` None``. A declaração acima é equivalente a

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

Você pode excluir registros por id:

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

e isto é equivalente a

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

e exclui o registro com o dado `` id``, se ele existir.

Nota: esta sintaxe de atalho de exclusão actualmente não trabalha se \* versionamento \* é ativado

Você pode inserir registros:

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

É equivalente a

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

e cria um novo registro com valores de campos especificados pelo dicionário sobre o lado direito.

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

Você pode atualizar os registros:

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

o qual é equivalente a

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

e atualiza um registro existente com os valores dos campos especificados pelo dicionário sobre o lado direito.

### 7.9.4 A obtenção de um `` row``

No entanto, outra sintaxe conveniente é o seguinte:

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

Aparentemente semelhante a ``db.mytable[id]`` a sintaxe acima é mais flexível e mais seguro. Antes de tudo, verifica se ``id`` é um int (ou ``str(id)`` é um int) e retorna ``None`` se não (nunca levanta uma exceção). Ele também permite especificar várias condições que o registro deve atender. Se eles não forem atendidas, ele também retorna ``None``.

### 7.9.5 Recursivas ``s SELECT``

Considere a pessoa tabela anterior e uma nova tabela “coisa” fazendo referência a uma “pessoa”:

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

e um simples selecionar a partir desta tabela:

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

o qual é equivalente a

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

onde ``\_id`` é uma referência para a chave principal da tabela. Normalmente ``db.thing.\_id`` é o mesmo que ``db.thing.id`` e vamos supor que na maior parte deste livro.

Para cada linha de coisas é possível buscar não apenas campos da tabela selecionada (coisa), mas também a partir de tabelas vinculadas (de forma recursiva):

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

Aqui ``thing.owner\_id.name`` requer um banco de dados escolha para cada coisa em coisas e por isso é ineficiente. Sugerimos usando junta sempre que possível, em vez de seleciona recursiva, no entanto, este é conveniente e prático ao acessar registros individuais.

Você também pode fazê-lo para trás, escolhendo os coisas referenciados por uma pessoa:

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

Nesta última expressão ``person.thing`` é um atalho para

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

isto é, o conjunto de thing`` ``s referenciados pelos actuais ``Person``. Esta sintaxe se decompõe se a tabela referenciando tem várias referências à tabela referenciada. Neste case é preciso ser mais explícito e usar uma consulta completa.

### 7.9.6 ``OrderBy``, ``groupby``, ``limitby``, ``distinct``, ``having``, ``orderby\_on\_limitby``, ``join``, ``left``, ``cache``

O comando ``SELECT`` leva uma série de argumentos opcionais.

#### ordenar por

Você pode buscar os registros classificados pelo nome:

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



```
Alex
Bob
Carl
```

Você pode buscar os registros classificados pelo nome em ordem inversa (aviso o til):

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

Você pode ter os registros obtida aparecem em ordem aleatória:

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

O uso de `` orderby = "<random>" `` não é suportada no Google NoSQL. No entanto, para superar esse limite, a classificação pode ser feito em linhas selecionadas:

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

Você pode classificar os registros de acordo com vários campos concatenando-os com um "|" :

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

### groupby, tendo

Usando `` groupby`` juntamente com `` orderby``, você pode agrupar registros com o mesmo valor para o campo especificado (isto é back-end específico, e não é sobre o Google NoSQL):

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

Pode usar `` having`` em conjunto com `` groupby`` ao grupo condicionalmente (apenas aqueles `` having`` a condição estão agrupados).

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

Observe que os registros filtros Consulta1 a ser exibido, registros filtros Query2 ser agrupados.

### distinto

Com o argumento `` distinta = True``, você pode especificar que você só quer selecionar registros distintos. Isto tem o mesmo efeito que o agrupamento usando todos os campos especificados, exceto que ele não necessita de classificação. Ao usar distinta é importante não para selecionar todos os campos, e em particular para não selecionar o campo "id", senão todos os registros serão sempre distintas.

Aqui está um exemplo:

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

Note que ``distinct`` também pode ser uma expressão, por exemplo:

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

### limitby

Com ``limitby = (min, max)`` , pode seleccionar um subconjunto dos registos de deslocamento = min, mas não incluindo offset = máx. No próximo exemplo nós seleccionamos os dois primeiros registos a partir de zero:

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

### orderby\_on\_limitby

Note-se que os padrões DAL de adicionar implicitamente um orderby ao usar um limitby. Isso garante a mesma consulta retorna os mesmos resultados de cada vez, importante para a paginação. Mas pode causar problemas de desempenho. use ``orderby\_on\_limitby = False`` para mudar isso (isso o padrão é True).

### juntar-se, deixou

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, em cache

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 Operadores lógicos

As consultas podem ser combinados usando o binário operador AND ``&``:

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

eo binário operador OR ``|``:

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

Você pode negar uma sub-consulta invertendo o seu operador:

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

ou pela negação explícita com o “~” operador unário:

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

Devido a restrições de Python em sobrecarga “and” e “or” operadores, estes não podem ser utilizados na formação de consultas. Os operadores binários “&” e “|” *deve ser usado em seu lugar*. Note-se que estes operadores (ao contrário de “and” e “or”) tem precedência maior do que os operadores de comparação, de modo que os parênteses “extra” nos exemplos acima são de preenchimento obrigatório. Da mesma forma, o operador unitário “~” *tem precedência mais elevada do que os operadores de comparação*, de modo ~ “comparações -negated também deve estar entre parênteses.

Também é possível consultas construir usando in-place operadores lógicos:

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

## 7.9.8 “Count”, “isempty”, “DELETE”, “update”

Você pode contar registros em um conjunto:

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

Note que “count” leva um opcional “distinct” argumento que o padrão é falso, e ele funciona muito parecido com o mesmo argumento para “SELECT”. “Count” tem também um argumento “cache” que funciona muito parecido com o argumento equivalente do método *SELECT*.

Às vezes você pode precisar verificar se uma tabela está vazia. Uma maneira mais eficiente do que a contagem está a utilizar o método *isempty*:

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

Você pode excluir registros em um jogo:

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

A “DELETE” método devolve o número de registros que foram eliminados.

E você pode atualizar todos os registros em um conjunto, passando argumentos nomeados correspondentes aos campos que precisam ser atualizados:

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

O método *update* retorna o número de registros que foram atualizados.

## 7.9.9 Expressões

O valor atribuído uma instrução de atualização pode ser uma expressão. Por exemplo, considere este modelo

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

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

Os valores usados em consultas também podem ser expressões

```
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``

Uma expressão pode conter uma cláusula case, por exemplo:

```
>>> 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 também permite actualizar um único registro que já está na memória usando ``update\_record``

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

``Update\_record`` não deve ser confundido com

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

porque para uma única linha, o método ``update`` atualiza o objeto de linha, mas não o registro de banco de dados, como no case de ``update\_record``.

Também é possível alterar os atributos de uma linha (um de cada vez) e, em seguida, chamar ``update\_record()`` sem argumentos para salvar as alterações:

```
>>> 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`.

O método `update_record` está disponível apenas se campo `id` da tabela está incluído no seletor, e `cacheable` não está definido para `True`.

### 7.9.12 Inserir e atualizar a partir de um dicionário

Um problema comum é composto de precisar inserir ou atualizar registros em uma tabela onde o nome da tabela, o campo para ser atualizado, eo valor para o campo são armazenados em variáveis. Por exemplo: ``tablename``, ``fieldname``, e ``value``.

A inserção pode ser feito usando a seguinte sintaxe:

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

A atualização do registro com dado id pode ser feito com:

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

Observe que usamos ``table.\_id`` ao invés de ``table.id``. Desta forma, a consulta funciona mesmo para tabelas com um campo de chave primária com o tipo diferente de "id".

### 7.9.13 `` `` First`` e last``

Dado um objecto linhas contendo registros:

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

são equivalentes às

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

Observe, `` primeiro () `` e `` última () `` permitem obter, obviamente, o primeiro e último registro presente em sua consulta, mas isso não significa que esses registros estão indo para ser o primeiro ou o último inserido registros. No caso de pretender o primeiro ou último registro inserido em uma determinada tabela não se esqueça de usar `` orderby = db.table\_name.id``. Se você esquecer você só vai conseguir o primeiro eo último registro retornado pela consulta, que são muitas vezes em uma ordem aleatória determinada pelo otimizador de consulta backend.

### 7.9.14 `` `` As\_dict`` e as\_list``

Fila objecto pode ser serializados em um dicionário normal usando a `` as\_dict () `` método e um objecto de linhas pode ser serializados em uma lista de dicionários usando a `` as\_list () `` método. aqui estão alguns exemplos:

```
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 connection):

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

### 7.9.15 Combinando Rows

Fileiras objects podem ser combinadas no nível Python. Aqui assumimos:

```
>>> print(rows1)
person.name
Max
```

```
Tim

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

Você pode fazer a união dos registros em dois conjuntos de linhas:

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

Você pode fazer a união dos registros remoção de duplicatas:

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

Você pode fazer intersecção dos registros em dois conjuntos de linhas:

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

### 7.9.16 ``Find``, ``exclude``, ``sort``

Algumas vezes você precisa executar duas seleções e um contém um subconjunto de um seletor anterior. Neste caso, é inútil para acessar o banco de dados novamente. Os ``find``, ``exclude`` e ``sort`` permitem manipular fileiras objeto e gerar outro sem acessar o banco de dados. Mais especificamente: - ``return find`` um novo conjunto de linhas filtradas por uma condição e deixa o inalterados originais. - ``return exclude`` um novo conjunto de linhas filtrados por uma condição e remove-os das linhas originais. - ``return sort`` um novo conjunto de linhas classificadas por uma condição e deixa o inalterados originais.

Todos estes métodos dar um único argumento, uma função que age em cada linha individual.

Aqui está um exemplo de uso:

```
>>> 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

```

Eles podem ser combinados:

```

>>> 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

```

Tipo leva um argumento opcional `reversa = True` com o significado óbvio.

O método `find` tem um argumento `limitby` opcional com a mesma sintaxe e funcionalidade como o conjunto `método SELECT`.

### 7.9.17 Selects com cache

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.

No exemplo a seguir, você vê um controlador que armazena em cache um seletor sobre a mesa `db.log` previamente definido. As buscas reais dados selecionados do banco de dados back-end não mais do que uma vez a cada 60 segundos e armazena o resultado na memória. Se a próxima chamada para este controlador ocorre em menos de 60 segundos desde o último banco de dados IO, ele simplesmente vai buscar os dados anteriores da memória.

```

def cache_db_select():
 logs = db().select(db.log.ALL, cache=(cache.get, 60))
 return dict(logs=logs)

```

O método `SELECT` tem um argumento `cacheable` opcional, normalmente definido como `False`. Quando `cacheável = True` o resultante `Rows` Serializável mas `A falta row` s `update_record` e métodos `delete_record`.

Se você não precisar destes métodos você pode acelerar seleciona um lote, definindo o atributo `cacheable`:

```
rows = db(query).select(cacheable=True)
```

Quando o argumento `cache` está definido, mas `cacheable = False` (default), apenas os resultados de banco de dados são armazenados em cache, não as linhas reais objeto. Quando o argumento `cache` é usado em conjunto com `cacheável = True` as linhas inteiras objecto é cache e isso resulta em muito mais rápido cache:

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

## 7.10 Computed and Virtual fields

### 7.10.1 Campos computados

Campos DAL podem ter um atributo ``compute``. Esta deve ser uma função (ou lambda) que recebe um objeto Row e retorna um valor para o campo. Quando um novo registro é modificado, incluindo inserções e atualizações, se um valor para o campo não é fornecido, py4web tenta calcular a partir dos outros valores de campo utilizando a função ``compute``. Aqui está um exemplo:

```
>>> 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 Campos virtuais

Campos virtuais também são computados campos (como na subseção anterior), mas eles diferem daquelas porque são **virtual** no sentido de que não são armazenadas no db e eles são calculados a cada vez registros são extraídos do banco de dados. Eles podem ser usados para simplificar o código do usuário sem usar armazenamento adicional, mas eles não podem ser usados para pesquisa.

### 7.10.3 Campos virtuais novo estilo (experimental)

py4web fornece uma nova e mais fácil maneira de definir campos virtuais e campos virtuais preguiçosos. Esta seção é marcado experimental porque as APIs ainda podem mudar um pouco do que é descrito aqui.

Aqui vamos considerar o mesmo exemplo na subseção anterior. Em particular, considere o seguinte modelo:

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

Pode-se definir um ``total\_price`` campo virtual como

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

isto é, simplesmente definindo um novo campo ``total\_price`` ser um ``Field.Virtual``. O único argumento do construtor é uma função que recebe uma linha e retorna os valores calculados.

Um campo virtual definido como o descrito acima é calculado automaticamente para todos os registros quando os registros são selecionados:



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

Também é possível definir campos de métodos que são calculados on-demand, quando chamado. Por exemplo:

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

Neste case, ``row.discounted\_total`` não é um valor, mas uma função. A função usa os mesmos argumentos que a função passada para o ``construtor Method`` exceto ``row`` que está implícito (pense nisso como ``self`` para objetos).

O campo preguiçoso no exemplo acima permite uma para calcular o valor total para cada ``item``:

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

E também permite passar um ``percentual discount`` opcional (digamos 15%):

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

Campos virtuais e de método também podem ser definidos no lugar quando uma tabela é definida:

```
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 Campos virtuais velho antigo

A fim de definir um ou mais virtuais campos, você também pode definir uma classe de contêiner, instanciá-lo e vinculá-lo a uma tabela ou a um seletor. Por exemplo, considere a seguinte tabela:

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

Pode-se definir um ``total\_price`` campo virtual como

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

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

Observe que cada método da classe que recebe um único argumento (auto) é um novo campo virtual. ``Self`` refere-se a cada linha de uma select. valores de campo são referidos pelo caminho completo como em ``self.item.unit\_price``. A tabela está ligada aos campos virtuais anexando uma instância da classe para atributo ``virtualfields`` da tabela.

Campos virtuais também podem acessar campos recursivos como em

```
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())
```

Observe o acesso de campo recursiva ``self.order\_item.item.unit\_price`` onde ``self`` é o registro looping.

Eles também podem agir sobre o resultado de um 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)
```

Note como neste case, a sintaxe é diferente. O campo virtual acessa tanto ``self.item.unit\_price`` e ``self.order\_item.quantity`` que pertencem ao juntar-se selecionar. O campo virtual é anexado para as linhas da tabela usando o método *setvirtualfields* do objecto linhas. Este método leva um número arbitrário de argumentos nomeados e pode ser usado para definir vários campos virtuais, definidos em várias classes, e anexá-los a várias tabelas:

```
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)
```

Campos virtuais podem ser *\* lazy\** ; tudo que eles precisam fazer é retornar uma função e acessá-lo chamando a função:

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

class MyVirtualFields:
 def lazy_total_price(self):
 def lazy(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())
```

ou mais curto utilizando uma função lambda:

```
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 Um para muitos relação

Para ilustrar como implementar um para muitos relação com a DAL, definir outra mesa “coisa” que refere-se à mesa “pessoa” que redefinir aqui:

```
>>> 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)`.

Este último é sempre convertido para o ex. Eles são equivalentes, exceto no case de tabelas preguiçosos, referências auto ou outros tipos de referências cíclicas onde o ex-notação é a notação só é permitido.

Agora, insira três coisas, duas de propriedade de Alex e um por 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
```

Você pode selecionar como você fez para qualquer outra tabela:

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

Porque uma coisa tem uma referência a uma pessoa, uma pessoa pode ter muitas coisas, assim que um registro da tabela pessoa agora adquire uma coisa nova atributo, que é um conjunto, que define as coisas dessa pessoa. Isso permite que um loop sobre todas as pessoas e buscar as suas coisas com facilidade:

```
>>> 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

Outra forma de conseguir um resultado semelhante é usando uma junção, especificamente um INNER JOIN. executa py4web junta-se automaticamente e de forma transparente quando a consulta liga dois ou mais tabelas como no exemplo a seguir:

```
>>> 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 que py4web fez uma junção, então as linhas agora contêm dois registros, um de cada mesa, ligados entre si. Porque os dois registros podem ter campos com nomes conflitantes, você precisa especificar a tabela quando se extrai um valor de campo de uma linha. Isto significa que enquanto antes que você poderia fazer:

```
row.name
```

e era óbvio que se este era o nome de uma pessoa ou uma coisa, em resultado de uma junção que você tem que ser mais explícito e dizer:

```
row.person.name
```

ou:

```
row.thing.name
```

Há uma sintaxe alternativa para associações internas:

```
>>> 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
```

Enquanto a saída é o mesmo, o SQL gerado nos dois cases, pode ser diferente. O último sintaxe remove as ambiguidades possíveis quando a mesma tabela é unidas duas vezes e alias:

```
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)])
```

O valor de ``join`` pode ser lista de ``db.table.on(...)`` para participar.

### 7.11.3 Left outer join

Observe que Carl não aparece na lista acima, porque ele não tem as coisas. Se você pretende selecionar sobre as pessoas (se eles têm coisas ou não) e as suas coisas (se tiver algum), então você precisa para realizar um LEFT OUTER JOIN. Isso é feito usando o argumento de “esquerda” da seleção. Aqui está um exemplo:

```
>>> 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(...)
```

é que a esquerda se juntar a consulta. Aqui o argumento de ``db.thing.on`` é a condição necessária para a junção (o mesmo utilizado acima para a junção interna). No case de uma associação à esquerda, é necessário ser explícito sobre quais campos para selecionar.

Multiple esquerda junções podem ser combinados, passando uma lista ou tupla de ``db.mytable.on(...)`` para o parâmetro ``left``.

### 7.11.4 Agrupamento e contando

Ao fazer junta-se, às vezes você quer agrupar linhas de acordo com certos critérios e contá-los. Por exemplo, contar o número de coisas pertencentes a cada pessoa. py4web permite isso também. Primeiro, você precisa de um operador de contagem. Em segundo lugar, você quer se juntar a tabela a pessoa com o quadro de coisa pelo proprietário. Terceiro, você quer selecionar todas as linhas (pessoa + coisa), agrupá-los por pessoa, e contá-los enquanto agrupamento:

```
>>> 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
```

Observe a ``operador count`` (que é incorporado) é usado como um campo. O único problema aqui é em como recuperar a informação. Cada linha contém claramente uma pessoa e a contagem, mas a contagem não é um campo de uma pessoa nem é uma mesa. Então, onde ela vai? Ele vai para o objeto de armazenamento representando o registro com uma chave igual ao próprio expressão de consulta.

O método `count` do objeto campo tem um argumento `distinct` opcional. Quando ajustado para ``True`` especifica que apenas os valores distintos de campo em questão estão a ser contadas.

### 7.11.5 Many to many relation

Nos exemplos anteriores, que permitiram uma coisa para ter um proprietário, mas uma pessoa pode ter muitas coisas. E se barco era propriedade de Alex e Curt? Isso requer uma relação muitos-para-muitos, e é realizada através de uma tabela intermediária que liga uma pessoa a uma coisa através de uma relação de propriedade.

Aqui está como fazê-lo:

```
>>> 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)>
```

a relação de propriedade existente pode agora ser reescrita como:

```
>>> 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
```

Agora você pode adicionar a nova relação que Curt co-proprietária Barco:

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

Porque agora você tem uma relação de três vias entre as mesas, pode ser conveniente para definir um novo conjunto no qual executar as operações:

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

Agora é fácil para selecionar todas as pessoas e suas coisas da nova 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
```

Da mesma forma, você pode procurar por todas as coisas pertencentes a Alex:

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

e todos os proprietários de barco:

```
>>> 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 A auto-referência e aliases

É possível definir tabelas com campos que se referem a si mesmos, aqui está um exemplo:

```
db.define_table('person',
 Field('name'),
 Field('father_id', 'reference person'),
 Field('mother_id', 'reference person'))
```

Observe que a notação alternativa de usar um objeto de tabela como tipo de campo irá falhar neste case, porque ele usa uma tabela antes de ser definido:

```
db.define_table('person',
 Field('name'),
 Field('father_id', db.person), # wrong!
 Field('mother_id', db['person'])) # wrong!
```

Em geral ``db.tablename`` e ``referência tablename`` são tipos de campo equivalentes, mas o último é o único que tem permissão para auto-referências.

Quando uma tabela tem uma auto-referência e você tem que fazer se juntar, por exemplo, para selecionar uma pessoa e seu pai, você precisa de um alias para a tabela. Em SQL um alias é um nome alternativo temporário que você pode usar para fazer referência a uma tabela / coluna em uma consulta (ou outra instrução SQL).

Com py4web você pode fazer um alias para uma tabela usando o método `with_alias`. Isso funciona também para expressões, o que significa também para campos desde ``Field`` é derivada de ``Expression``.

Aqui está um exemplo:

```
>>> 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
```

Observe que optamos por fazer uma distinção entre: - “father\_id”: o nome do campo usado na “pessoa” mesa; - “pai”: o alias que deseja usar para a tabela referenciada pelo campo acima; esta é comunicada ao banco de dados; - “Pai”: a variável usada por py4web para se referir a esse alias.

A diferença é sutil, e não há nada de errado em usar o mesmo nome para os três:

```
>>> 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

```

Mas é importante ter a distinção clara, a fim de construir perguntas corretas.

## 7.12 Outros operadores

py4web tem outros operadores que fornecem uma API para operadores SQL equivalentes de acesso. Vamos definir outra mesa “log” para eventos loja de segurança, sua event\_time e gravidade, onde a gravidade é um número inteiro.

```

>>> db.define_table('log', Field('event'),
... Field('event_time', 'datetime'),
... Field('severity', 'integer'))
<Table log (id, event, event_time, severity)>

```

Como antes, inserir alguns eventos, a “varredura de portas”, uma “injeção de XSS” e um “login não autorizado”. Por causa do exemplo, você pode registrar eventos com o mesmo event\_time mas com diferentes gravidades (1, 2, e 3, respectivamente).

```

>>> 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``

Campos tem um `` operador like`` que você pode usar para combinar strings:

```

>>> for row in db(db.log.event.like('port%')).select():
... print(row.event)
...
port scan

```

Aqui “porta%” indica uma partida string com “porta”. O personagem por cento sinal, “%”, é um personagem wild-card que significa “qualquer sequência de caracteres”.

O `` operador like`` mapeia para a palavra como em ANSI-SQL. COMO é sensível a maiúsculas na maioria dos bancos de dados, e depende do agrupamento do próprio banco de dados. O método *like* é, portanto, case-sensível, mas ele pode ser feito de maiúsculas e minúsculas com

```
db.mytable.myfield.like('value', case_sensitive=False)
```

que é o mesmo que usar `` ilike``

```
db.mytable.myfield.ilike('value')
```



py4web também fornece alguns atalhos:

```
db.mytable.myfield.startswith('value')
db.mytable.myfield.endswith('value')
db.mytable.myfield.contains('value')
```

que são aproximadamente equivalentes, respectivamente, a

```
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](#).

O método `contains` também pode ser passada uma lista de valores e um argumento booleano opcional `all` para procurar registros que contêm todos os valores:

```
db.mytable.myfield.contains(['value1', 'value2'], all=True)
```

ou qualquer valor a partir da lista

```
db.mytable.myfield.contains(['value1', 'value2'], all=False)
```

Há um também um `` método `regex``` que funciona como o método `like` mas permite sintaxe de expressão regular para a expressão look-up. Ele só é suportado pelo MySQL, Oracle, PostgreSQL, SQLite, e MongoDB (com diferente grau de apoio).

O `` `upper``` e `` `lower``` permitem converter o valor do campo para maiúsculas ou minúsculas, e você também pode combiná-los com o gosto do operador:

```
>>> 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```

A data e a data e hora campos têm `` `day```, `` `month``` e `` `year```. Os campos de data e hora e de tempo têm `` `hour```, `` `minutes``` e métodos `seconds```. Aqui está um exemplo:

```
>>> for row in db(db.log.event_time.year() > 2018).select():
... print(row.event)
...
port scan
xss injection
unauthorized login
```

### 7.12.3 `` `Belongs```

O operador IN SQL é realizado através do método `belongs` que devolve verdadeiro quando o valor do campo pertence ao conjunto especificado (lista ou tuplos):

```
>>> for row in db(db.log.severity.belongs((1, 2))).select():
... print(row.event)
...
port scan
xss injection
```

A DAL também permite que um SELECT aninhada como o argumento do operador `pertence`. A única limitação é que o seletor aninhada tem de ser um `` `_select```, não um `` `SELECT```, e apenas um campo tem de ser seleccionada explicitamente, o que define o conjunto.

```
>>> 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
```

Nos cases em que um seletor aninhado é necessária e o campo look-up é uma referência também podemos usar uma consulta como argumento. Por exemplo:

```
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()
```

Neste case, é óbvio que o SELECT aninhado só precisa do campo referenciado pelo campo ``db.thing.owner\_id`` por isso não precisa do ``notação mais detalhado \_select``.

A selecção pode aninhada também ser usado como insert valor / atualização, mas, neste case, a sintaxe é diferente:

```
lazy = db(db.person.name == 'Jonathan').nested_select(db.person.id)

db(db.thing.id == 1).update(owner_id = lazy)
```

Neste case, ``lazy`` é uma expressão aninhada que calcula o ``id`` de pessoa "Jonathan". As duas linhas resultar em uma consulta SQL única.

#### 7.12.4 ``Sum``, ``avg``, ``min``, ``max`` e ``len``

Anteriormente, você usou o ``operador count`` para contar registros. Da mesma forma, você pode usar o ``operador sum`` para adicionar (soma) os valores de um campo específico de um grupo de registros. Tal como no case de contagem, o resultado de uma soma é recuperado através do objecto de armazenamento:

```
>>> sum = db.log.severity.sum()
>>> print(db().select(sum).first()[sum])
6
```

Você também pode usar ``avg``, ``min``, e ``max`` à média, mínimo e valor máximo, respectivamente, para os registros selecionados. Por exemplo:

```
>>> max = db.log.severity.max()
>>> print(db().select(max).first()[max])
3
```

``Len`` calcula o comprimento do valor do campo. Ele é geralmente usado em cordas ou texto campos, mas dependendo do back-end que ainda pode funcionar para outros tipos também (boolean, integer, etc).

```
>>> for row in db(db.log.event.len() > 13).select():
... print(row.event)
...
unauthorized login
```

As expressões podem ser combinados para formar expressões mais complexas. Por exemplo, aqui estamos calculando a soma do comprimento das strings de eventos nos logs de mais um:

```
>>> exp = (db.log.event.len() + 1).sum()
>>> db().select(exp).first()[exp]
```

## 7.12.5 Substrings

Pode-se construir uma expressão para se referir a uma substring. Por exemplo, podemos agrupar as coisas cujo nome começa com os mesmos três personagens e selecione apenas um de cada grupo:

```
db(db.thing).select(distinct = db.thing.name[:3])
```

## 7.12.6 Os valores por defeito com `` coalesce`` e coalesce\_zero``

Há momentos em que você precisa para puxar um valor de banco de dados, mas também precisa de valores padrão se o valor para um registro é definido como NULL. Em SQL existe uma função, ``COALESCE``, para isso. py4web tem um método *coalesce* equivalente:

```
>>> 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
```

Outras vezes você precisa para calcular uma expressão matemática, mas alguns campos têm um valor definido para Nenhum quando deveria ser zero. ``Coalesce\_zero`` vem para o resgate por falta Nada a zero na consulta:

```
>>> 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 Exportar e importar dados

### 7.13.1 CSV (uma tabela de cada vez)

Quando um objeto linhas é convertido para uma string é automaticamente serializado na 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
```

Você pode serializar uma única tabela em formato CSV e armazená-lo em um arquivo “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()))
```

Isto é equivalente a

```
rows = db(db.person).select()
with open('test.csv', 'wb') as dumpfile:
 rows.export_to_csv_file(dumpfile)
```

Você pode ler o arquivo de volta CSV com:

```
with open('test.csv', 'rb') as dumpfile:
 db.person.import_from_csv_file(dumpfile)
```

Again, you can be explicit 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)
```

ea importação de um:

```
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 (todas as tabelas ao mesmo tempo)

Em py4web, você pode backup / restaurar um banco de dados inteiro com dois comandos:

Exportar:

```
with open('somefile.csv', 'w', encoding='utf-8', newline='') as dumpfile:
 db.export_to_csv_file(dumpfile)
```

Importar:

```
with open('somefile.csv', 'r', encoding='utf-8', newline='') as dumpfile:
 db.import_from_csv_file(dumpfile)
```

Este mecanismo pode ser utilizado mesmo se a banco de dados de importação é de um tipo diferente do que a banco de dados de exportação.

Os dados são armazenados em “somefile.csv” como um arquivo CSV, onde cada mesa começa com uma linha que indica o nome da tabela, e outra linha com os nomes de campos:

```
TABLE tablename
field1,field2,field3,...
```

Duas tabelas são separados por `` r n r n`` (que é duas linhas vazias). As extremidades de arquivos com a linha

```
END
```

O arquivo não inclui os arquivos enviados, se estes não são armazenados no banco de dados. Os

upload de arquivos armazenados no sistema de arquivos deve ser despejado em separado, um zip dos “uploads” pasta pode ser suficiente na maioria dos cases.

Ao importar, os novos registros serão anexados ao banco de dados se não está vazio. Em geral, os novos registros importados não terão o mesmo ID de registro como os registros originais (salvos), mas py4web irá restaurar referências para que eles não estão quebrados, mesmo que os valores id podem mudar.

Se uma tabela contém um campo chamado `` uuid``, este campo será utilizado para identificar duplicatas. Além disso, se um registro importado tem o mesmo `` uuid`` como um registro existente, o recorde anterior será atualizada.

### 7.13.3 CSV e sincronização de banco de dados remoto

Considere mais uma vez o seguinte modelo:

```
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)
```

Cada registro é identificado por um identificador e referenciado por esse id. Se você tem duas cópias do banco de dados usado por instalações py4web distintas, o id é único apenas dentro de cada banco de dados e não através das bases de dados. Este é um problema ao mesclar registros de bancos de dados diferentes.

A fim de fazer registros exclusivamente identificável através de bases de dados, eles devem: - ter um ID único (UUID), - ter uma última modificação para acompanhar o mais recente entre várias cópias, - referência o UUID em vez do id.

Isto pode ser conseguido mudando o modelo acima para:

```
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)
```

Note-se que nas definições da tabela acima, o valor padrão para os dois campos `` uuid`` é definida como uma função de lambda, que retorna um UUID (convertido para uma strings). A função lambda é chamado uma vez para cada registro inserido, garantindo que cada registro receba um UUID único, mesmo que vários registros são inseridos em uma única transação.

Criar uma ação de controlador para exportar o banco de dados:

```
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()
```

Criar uma ação de controlador para importar uma cópia salva dos outros registros de dados e sincronização:

```
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)
```

Opcionalmente, você deve criar um índice manualmente para fazer a busca por uuid mais rápido.

Alternativamente, você pode usar XML-RPC para exportar / importar o arquivo.

Se os registros referência a arquivos enviados, você também precisa exportar / importar o conteúdo da pasta uploads. Observe que os arquivos nele já são rotulados por UUIDs para que você não precisa se preocupar com conflitos de nomes e referências.

### 7.13.4 HTML e XML (uma tabela de cada vez)

Linhas objetos também têm um método `xml` (como ajudantes) que serializa-lo para XML / HTML:

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

```
<table>
<thead>
<tr><th>person.id</th><th>person.name</th>
 <th>thing.id</th><th>thing.name</th>
 <th>thing.owner_id</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 Representação de dados

O método `Rows.export_to_csv_file` aceita um argumento de palavra-chave chamada `represent`. Quando `True` ele usará as colunas função `represent` ao exportar os dados, em vez dos dados brutos.

A função também aceita um argumento de palavra-chave chamada `colnames` que deve conter uma lista de nomes de colunas um desejo para exportação. O padrão é todas as colunas.

Ambos `export_to_csv_file` e `import_from_csv_file` aceitar argumentos de palavra-chave que contam o analisador CSV o formato para salvar / carregar os arquivos: - `delimiter`: delimitador para separar valores (padrão `","`) - `quotechar`: personagem para usar para citar valores String (default para aspas) - `quoting`: sistema de cotação (padrão `csv.QUOTE_MINIMAL`)

Aqui estão algumas Exemplo de uso:

```
import csv
rows = db(query).select()
with open('/tmp/test.txt', 'wb') as outfile:
 rows.export_to_csv_file(outfile,
 delimiter='|',
 quotechar='"',
 quoting=csv.QUOTE_NONNUMERIC)
```

O que tornaria algo semelhante a

```
"hello"|35|"this is the text description"|"2013-03-03"
```

Para mais informações consulte a documentação oficial do Python

## 7.14 Características avançadas

### 7.14.1 `` Lista: <type> `` e `` contains``

py4web fornece os seguintes tipos de campos especiais:

```
list:string
list:integer
list:reference <table>
```

Eles podem conter listas de cordas, de inteiros e de referências, respectivamente.

No Google App Engine NoSQL `` lista: string`` é mapeado em `` StringListProperty``, os outros dois são mapeados em `` ListProperty (int) ``. Em bancos de dados relacionais são mapeados em campos de texto que contém a lista de itens separados por `` | ``. Por exemplo `` [1, 2, 3] `` é mapeado para `` 1 | 2 | 3 | ``.

Para listas de corda os itens são escapou de modo que qualquer `` | `` no item é substituído por um `` || ``. De qualquer forma esta é uma representação interna e é transparente para o usuário.

Você pode usar `` lista: string``, por exemplo, da seguinte maneira:

```
>>> 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']
```

`` Lista: obras integer`` da mesma forma, mas os itens devem ser inteiros.

Como de costume, os requisitos são aplicadas ao nível das formas, não no nível de `` insert``.

Por `` lista: <type> `` campos de `` contém (valor) `` operador de mapas em uma consulta não trivial que verifica a existência de listas contendo o `` value``. O `` operador contains`` também funciona para regular, `` string`` e `` campos text`` e ele mapeia para um `` LIKE "% value%" ``.

O `` lista: reference`` eo `` contém (valor) `` operador são particularmente úteis para de-normalize muitos-para-muitos relações. Aqui está um exemplo:

```
>>> 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
```

Observe que um `` lista: Campo tag`` referência obter uma restrição padrão

```
requires = IS_IN_DB(db, db.tag._id, db.tag._format, multiple=True)
```

que produz um `` / OPTION`` gota-caixa múltipla SELECT formas.

Além disso, observe que este campo recebe um atributo *represent* que representa a lista de referências como uma lista separada por vírgulas de referências formatados padrão. Isto é usado em leitura `` forms``.

Enquanto `` lista: reference`` tem um validador padrão e uma representação padrão, `` lista: integer`` e `` lista: string`` não. Então, esses dois precisam de um `` IS\_IN\_SET`` ou um `` validador IS\_IN\_DB`` se você quiser usá-los em formas.

### 7.14.2 Herança de tabela

É possível criar uma tabela que contém todos os campos de outra tabela. É suficiente para passar a outra tabela no lugar de um campo para `` define\_table``. Por exemplo

```
>>> 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)>
```

Também é possível definir uma tabela fictícia que não está armazenado em um banco de dados, a fim de reutilizá-la em vários outros lugares. Por exemplo:

```
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)
```

Este exemplo parte do princípio que a autenticação py4web padrão está activada.

Note que se você usar `` Auth`` py4web já cria uma tal mesa para você:

```
auth = Auth(db)
db.define_table('payment', Field('amount', 'double'), auth.signature)
```

Ao usar herança de tabela, se você deseja que a tabela herdar a validadores herdar, certifique-se de definir os validadores de tabela pai antes de definir a tabela herdar.

### 7.14.3 `` `` Filter\_in`` e filter\_out``

É possível definir um filtro para cada campo a ser chamada antes de um valor é inserido na banco de dados para esse campo e depois de um valor é recuperado a partir da banco de dados.

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 retornos de chamada no registro de inserção, exclusão e atualização

PY4WEB fornece um mecanismo para registrar retornos de chamada para ser chamado antes e / ou após a inserção, atualização e exclusão de registros.

Cada tabela armazena seis listas de chamadas de retorno:

```

db.mytable._before_insert
db.mytable._after_insert
db.mytable._before_update
db.mytable._after_update
db.mytable._before_delete
db.mytable._after_delete

```

Você pode registrar uma função de retorno de chamada, acrescentando-o à lista correspondente. A ressalva é que, dependendo da funcionalidade, o retorno tem assinatura diferente.

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

```

Como você pode ver: - ``f`` é passado o objeto ``OpRow`` com os dados para inserção ou atualização. - ``i`` é passado o id do registro recém-inserido. - ``s`` é passado o objeto ``Set`` usado para atualizar ou excluir. ``OpRow`` é um objeto auxiliar especializada em armazenamento (campo, valor) pares, você pode pensar nisso como um dicionário normal que você pode usar até mesmo com a sintaxe da notação atributo (que é ``f.name`` e ``f["nome"]`` são equivalentes).

Os valores de retorno destes callback deve ser ``None`` ou ``False``. Se qualquer um dos ``\_antes\_\*

`` retorno de chamada retorna um `` valor True`` ele irá abortar a / update / operação de exclusão real de inserção.

Algumas vezes uma chamada de retorno pode precisar executar uma atualização na mesma ou em uma tabela diferente e se quer evitar disparar outras chamadas de retorno, o que poderia causar um loop infinito.

Para este efeito, há os objetos `` Set`` tem um método `update_naive` que funciona como `update` mas ignora antes e depois de retornos de chamada.

### Cascades no banco de dados

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 versionamento recorde

É possível pedir py4web para salvar cada cópia de um registro quando o registro é modificado individualmente. Existem diferentes maneiras de fazer isso e que pode ser feito para todas as tabelas ao mesmo tempo usando a sintaxe:

```
auth.enable_record_versioning(db)
```

isso requer `` Auth``. Ele também pode ser feito para cada mesa, como discutido abaixo.

Considere a seguinte tabela:

```
db.define_table('stored_item',
 Field('name'),
 Field('quantity', 'integer'),
 Field('is_active', 'boolean',
 writable=False, readable=False, default=True))
```

Observe o campo booleano oculto chamado `` is\_active`` e padronizando para True.

Podemos dizer py4web para criar uma nova tabela (no mesmo ou em outro banco de dados) e armazenar todas as versões anteriores de cada registro na tabela, quando modificado.

Isso é feito da seguinte maneira:

```
db.stored_item._enable_record_versioning()
```

ou em uma sintaxe mais detalhado:

```
db.stored_item._enable_record_versioning(archive_db=db,
 archive_name='stored_item_archive',
 current_record='current_record',
 is_active='is_active')
```

O `` archive\_db = db`` diz py4web para armazenar a tabela de arquivo no mesmo banco de dados como o `` tabela stored\_item``. O `` archive\_name`` define o nome para a tabela de arquivo. A tabela de arquivo tem os mesmos campos como a tabela original `` stored\_item`` exceto que campos exclusivos não são mais exclusivo (porque ele precisa para armazenar várias versões) e tem um campo extra que nome é especificado por `` current\_record`` e que é uma referência para o registro atual na tabela `` stored\_item``.

Quando os registros são excluídos, eles não são realmente excluídos. Um registro excluído é copiado na tabela `` stored\_item\_archive`` (como quando ele é modificado) e do campo `` is\_active`` é definido como False. Ao permitir gravar versões conjuntos py4web um `` common\_filter`` nesta tabela que esconde todos os registros na tabela `` stored\_item`` onde o campo `` is\_active`` é definida como falsa. O parâmetro `` is\_active`` no método `_enable_record_versioning` permite especificar o nome do campo usado pelo `common_filter` para determinar se o campo foi excluído ou não.

### 7.14.6 filtros comuns

Um filtro comum é uma generalização da ideia multi-tenancy acima. Ele fornece uma maneira fácil de evitar a repetição da mesma consulta. Considere, por exemplo, a tabela a seguir:

```
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)
```

Qualquer select, DELETE ou UPDATE nesta tabela, vai incluir posts única públicos. O atributo também pode ser modificado em tempo de execução:

```
db.blog_post._common_filter = lambda query: ...
```

Ela serve tanto como uma forma de evitar a repetição do “db.blog\_post.is\_public == True” frase em cada blog pesquisa post, e também como uma melhoria de segurança, que o impede de esquecer para não permitir a visualização de mensagens não-públicas.

No case de você realmente quer itens deixados de fora pelo filtro comum (por exemplo, permitindo que o administrador para ver mensagens não-públicas), você pode remover o filtro:

```
db.blog_post._common_filter = None
```

ou ignorá-lo:

```
db(query, ignore_common_filters=True)
```

Note-se que common\_filters são ignorados pela interface AppAdmin.

### 7.14.7 Personalizados ``tipos Field``

Além de usar o ``filter\_in`` e ``filter\_out``, é possível definir novos tipos de campos / personalizados. Por exemplo, suponha que você deseja definir um tipo personalizado para armazenar um endereço IP:

```
>>> 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`` é uma fábrica tipo de campo. Seu argumento ``type`` deve ser um dos tipos py4web padrão. Diz py4web como tratar os valores de campo no nível py4web. ``Native`` é o tipo do campo, tanto quanto a banco de dados está em causa. nomes permitidos dependem do mecanismo de banco de dados. ``Encoder`` é uma transformação opcional função aplicada quando os dados são armazenados e ``decoder`` é a função de transformação inversa opcional.

This feature is marked as experimental because can make your code not portable across database engines.

Ele não funciona no Google App Engine NoSQL.

### 7.14.8 Usando DAL sem definir tabelas

A DAL pode ser usado a partir de qualquer programa Python simplesmente fazendo isso:

```
from pydal import DAL, Field
db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases')
```

ou seja, importar a DAL, conexão e especificar a pasta que contém os arquivos .table (a pasta app / bancos de dados).

Para acessar os dados e seus atributos ainda temos que definir todas as tabelas que vão de acesso com ``db.define\_table``.

Se nós apenas precisamos de acesso aos dados, mas não para os atributos da tabela py4web, nós fugir sem re-definir as tabelas, mas simplesmente pedindo py4web para ler as informações necessárias a partir dos metadados nos ficheiros .table:

```
from py4web import DAL, Field
db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases',
auto_import=True)
```

Isso nos permite acessar qualquer db.table sem necessidade de re definir-lo.

### 7.14.9 Transação distribuída

No momento da escrita deste recurso só é suportado pelo PostgreSQL, MySQL e Firebird, uma vez que expõem API para commits de duas fases.

Supondo que você tenha dois (ou mais) conexões com bancos de dados PostgreSQL distintas, por exemplo:

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

Em seus modelos ou controladores, você pode cometê-los simultaneamente com:

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

Em case de falha, esta função desfaz e levanta uma ``Exception``.

Em controladores, quando uma ação retornos, se você tiver duas ligações distintas e você não chamar a função acima, py4web compromete-os separadamente. Isto significa que há uma possibilidade de

que um dos commits sucede e uma falha. A transação distribuída impede que isso aconteça.

### 7.14.10 Copiar dados de um para outro db

Considere a situação em que você estiver usando o seguinte banco de dados:

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

e você deseja mover para outro banco de dados usando uma sequência de conexão diferente:

```
db = DAL('postgres://username:password@localhost/mydb')
```

Antes de mudar, você quer mover os dados e reconstruir todos os metadados para o novo banco de dados. Assumimos o novo banco de dados a existir, mas nós também assumir que é vazio.

## 7.15 Pegadinhas

### 7.15.1 Nota sobre novo DAL e adaptadores

O código fonte do Banco de Dados Camada de Abstração foi completamente reescrito em 2010. Enquanto ele permanece compatível com versões anteriores, a reescrita tornou mais modular e mais fácil de estender. Aqui nós explicamos a lógica principal.

The module “dal.py” defines, among other, the following classes.

```
ConnectionPool
BaseAdapter extends ConnectionPool
Row
DAL
Reference
Table
Expression
Field
Query
Set
Rows
```

Seu uso tem sido explicado nas seções anteriores, exceto para ``BaseAdapter``. Quando os métodos de um ``Table`` ou ``necessidade objeto Set`` para se comunicar com o banco de dados que confiam aos métodos do adaptador a tarefa para gerar o SQL e ou a chamada de função.

Por exemplo:

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

chamadas

```
Table.insert(myfield='myvalue')
```

que delega o adaptador de voltar:

```
db._adapter.insert(db.mytable, db.mytable._listify(dict(myfield='myvalue')))
```

Aqui ``convertidos db.mytable.\_listify`` o dict dos argumentos em uma lista de `` (campo, valor) `` e chama o método *insert* do *adapter*. ``Db.\_adapter`` faz mais ou menos o seguinte:

```
query = db._adapter._insert(db.mytable, list_of_fields)
db._adapter.execute(query)
```

onde a primeira linha constrói a consulta e o segundo executa.

``BaseAdapter`` define a interface para todas as placas.

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)

```

que substituir o comportamento dos ``BaseAdapter``.

Cada adaptador tem mais ou menos a seguinte estrutura:

```

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])
```

Olhando para os vários adaptadores como exemplo deve ser fácil de escrever novos.

Quando ``db`` exemplo é criado:

```
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:psycpg2	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:psycpg2	pydal.adapters.postgres.PostgrePsycoNew
postgres3	pydal.adapters.postgres.PostgreBoolean
postgres3:psycpg2	pydal.adapters.postgres.PostgrePsycoBoolean
postgres:psycpg2	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 mysqlldb
from pydal.adapters.mysql import SQLAlchemyAdapter
SQLAlchemyAdapter.driver = mysqlldb
```

isto é ``mysqlldb`` tem de ser \* que \* módulo com um método .Connect (). Você pode especificar argumentos motorista opcionais e argumentos adaptador:

```
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 "mysqldb" 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 não tem um tipo booleano. py4web mapeia internamente booleans para uma strings de 1 carácter, com ‘T’ e ‘F’ representar Verdadeiro e Falso. A DAL lida com isso completamente; a abstração de um verdadeiro valor booleano funciona bem. Mas se você estiver atualizando a tabela SQLite com o SQL diretamente, estar ciente da implementação py4web, e evitar o uso de 0 e 1 valores.

## 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 be 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)

não MSSQL <2012 não suporta o SQL OFFSET palavra-chave. Portanto, o banco de dados não pode fazer a paginação. Ao fazer um ``limitby = (a, b)`` py4web vai buscar a primeira ``a + b`` linhas e descartar o primeiro ``a``. Isto pode resultar numa sobrecarga considerável quando comparado com outros bancos de dados. Se você estiver usando MSSQL >= 2005, o prefixo recomendado para uso é ``mssql3: //`` que fornece um método para evitar o problema de buscar todo o conjunto de resultados não-paginado. Se você estiver em MSSQL >= 2012, use ``mssql4: //`` que usa o ``OFFSET ... ROWS ... FETCH PRÓXIMO ... ROWS ONLY`` construção para apoiar a paginação nativamente, sem sucessos de desempenho como outros backends. O ``mssql: //`` uri também reforça (por razões históricas) o uso de ``colunas text``, que são superseeded em versões mais recentes (a partir de 2005) por ``varchar (max)``. ``Mssql3: //`` e ``mssql4: //`` deve ser usado se você não quer enfrentar algumas limitações do - oficialmente obsoleto - ``colunas text``.

MSSQL tem problemas com referências circulares em tabelas que têm onDelete CASCADE. Este é um bug MSSQL e você trabalhar em torno dele, definindo o atributo onDelete para todos os campos de referência a “nenhuma acção”. Você também pode fazê-lo uma vez por todas, antes de definir tabelas:

```
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 também tem problemas com argumentos passados para a palavra-chave DISTINCT e, portanto Enquanto isso funciona,

```
db(query).select(distinct=True)
```

isso não faz

```
db(query).select(distinct=db.mytable.myfield)
```

### 7.15.6 Oráculo

A Oracle também não suporta a paginação. Ele não suporta nem a OFFSET nem as palavras-chave limite. PY4WEB alcança a paginação, traduzindo um ``db (...). Select (limitby = (a, b))`` em um complexo de três vias SELECT aninhada (como sugerido por documentação oficial Oracle). Isso funciona para simples escolha, mas pode quebrar para seleciona complexos envolvendo campos e ou junta alias.

### 7.15.7 Google NoSQL (Datastore)

Google NoSQL (Datastore) não permite que se junta, deixou junta, agregados, expressão ou envolvendo mais de uma tabela, o ‘como’ pesquisas operador em campos “texto”.

As transações são limitados e não fornecida automaticamente pelo py4web (você precisa usar a API do Google ``run\_in\_transaction`` que você pode procurar na documentação do Google App Engine online).

O Google também limita o número de registros que você pode recuperar em cada uma consulta (1000, no momento da escrita). No Google armazenamento de dados IDs de registro são inteiro, mas eles não são seqüenciais. Enquanto em SQL “lista: string” tipo é mapeado em um tipo de “texto”, no Google Datastore é mapeado em um ``ListStringProperty``. Da mesma forma “lista: número inteiro” e “lista: referência” são mapeados para ``ListProperty``. Isso faz buscas por conteúdo dentro desses campos tipos mais eficientes no Google NoSQL que em bancos de dados SQL.

---

## 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”:

```
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](https://en.wikipedia.org/wiki/Representational_state_transfer)

[CIT0802] <https://graphql.org/>

```

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.

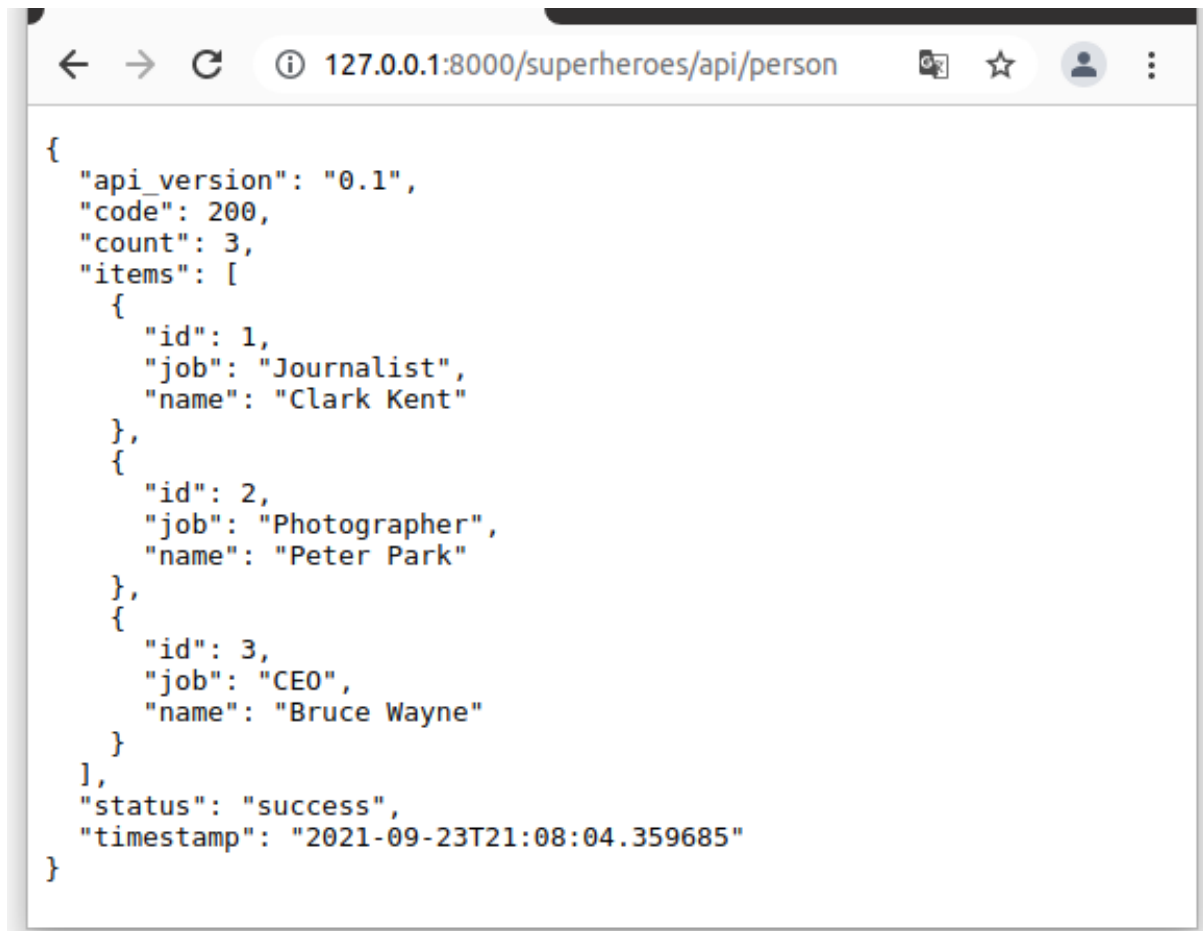
A acção acima referida é exposto como:

```

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

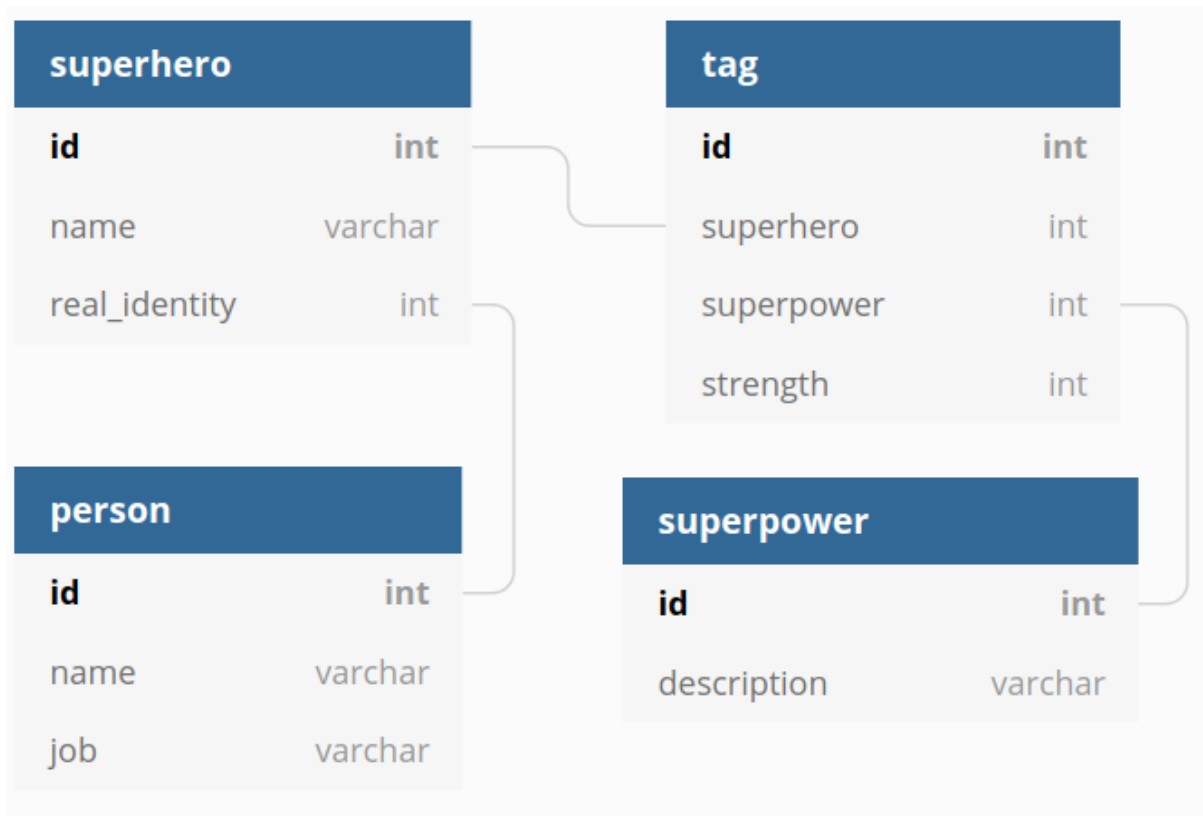
```

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

A screenshot of a web browser window. The address bar shows the URL '127.0.0.1:8000/superheroes/api/person'. The main content area displays a JSON object. The JSON structure includes an 'api\_version' of '0.1', a 'code' of 200, a 'count' of 3, and an 'items' array containing three objects. Each object represents a superhero with an 'id', a 'job', and a 'name'. The first item is Clark Kent (Journalist), the second is Peter Park (Photographer), and the third is Bruce Wayne (CEO). The JSON also includes a 'status' of 'success' and a 'timestamp' of '2021-09-23T21:08:04.359685'.

```
{
 "api_version": "0.1",
 "code": 200,
 "count": 3,
 "items": [
 {
 "id": 1,
 "job": "Journalist",
 "name": "Clark Kent"
 },
 {
 "id": 2,
 "job": "Photographer",
 "name": "Peter Park"
 },
 {
 "id": 3,
 "job": "CEO",
 "name": "Bruce Wayne"
 }
],
 "status": "success",
 "timestamp": "2021-09-23T21:08:04.359685"
}
```

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

A consulta geral tem a forma `` {} algo .eq = value`` onde `` eq = `` significa “igual”, `` gt = `` significa “maior que”, etc. A expressão pode ser prefixado por `` not``.

{something} can be:

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

**\*\* Todos os super-heróis chamado de “Superman” \*\***

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

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

**\*\* Todos os super-heróis com a identidade real “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:

**\*\* Todos os super-heróis com qualquer superpotência tag com força > 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:

**\*\* Todos os super-heróis com o poder de vôo \*\***

```
/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

Aqui estão alguns exemplos práticos:

URL:

```
/superheroes/api/superhero
```

RESULTADO:

```
{
 "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
```

RESULTADO:

```
{
```

```

"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
```

RESULTADO:

```
{
 "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
```

(Desnormalizar o real\_identity e renomeá-lo de identidade)

RESULTADO:

```
{
 "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]
```

(Desnormalizar o real\_identity [mas apenas campos nome e trabalho], recolher a com o prefixo de identidade)

RESULTADO:

```

{
 "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
```

RESULTADO:

```

{
 "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
```

RESULTADO:

```

{
 "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]

```

RESULTADO:

```

{
 "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]
```

RESULTADO:

```
{
 "count": 3,
 "status": "success",
 "code": 200,
 "items": [
 {
 "real_identity": 1,
 "name": "Superman",
 "powers": [
 {
 "strength": 100,
 "description": "Flight"
 },
 {
 "strength": 100,
 "description": "Strength"
 },
 {
 "strength": 100,
 "description": "Speed"
 },
 {
 "strength": 100,
 "description": "Durability"
 }
],
 "id": 1
 },
 {
 "real_identity": 2,
 "name": "Spiderman",
 "powers": [
 {
 "strength": 50,
 "description": "Strength"
 },
 {
 "strength": 75,
 "description": "Speed"
 },
 {
 "strength": 10,
 "description": "Durability"
 }
],
 "id": 2
 }
],
}
```



```

{
 "real_identity": 3,
 "name": "Batman",
 "powers": [
 {
 "strength": 80,
 "description": "Strength"
 },
 {
 "strength": 20,
 "description": "Speed"
 },
 {
 "strength": 70,
 "description": "Durability"
 }
],
 "id": 3
},
{
 "timestamp": "2019-05-19T05:38:00.355181",
 "api_version": "0.1"
}

```

URL (it's a single line, split for readability):

```

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

```

RESULTADO:

```

{
 "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
```

RESULTADO:

```
{
 "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
```

RESULTADO:

```
{
 "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
```

RESULTADO:

```
{
 "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
```

RESULTADO:

```
{
 "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.

---

## Linguagem de template YATL

---

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.

Aqui está um exemplo:

```
[[
if i == 0:
response.write('i is 0')
else:
response.write('i is not 0')
pass
]]
```

Note que ```pass``` é uma palavra-chave Python, não uma palavra-chave py4web. Alguns editores Python, como Emacs, use a palavra-chave ```pass``` para significar a divisão de blocos e usá-lo para o

código re-indent automaticamente.

O linguagem de template py4web faz exatamente a mesma. Quando encontra algo como:

```
<html><body>
[[for x in range(10):]][[=x]] hello
[[pass]]
</body></html>
```

que traduz em um programa:

```
response.write("""<html><body>""", escape=False)
for x in range(10):
 response.write(x)
 response.write(""" hello
""", 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).

Observe também que:

```
[[=x]]
```

gera

```
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).

Aqui está um exemplo que introduz o ``H1`` helper:

```
[[=H1(i)]]
```

que é traduzido para:

```
response.write(H1(i))
```

mediante avaliação, o objeto ``H1`` e seus componentes são recursivamente serializados, escapou e escrita para o corpo da resposta. As tags gerados pelo ``H1`` e HTML interior não escapamos. Este mecanismo garante que todo o texto - e somente texto - exibido na página web é sempre escaparam, evitando assim vulnerabilidades XSS. Ao mesmo tempo, o código é simples e fácil de depurar.

O método ``response.write(obj, escapar = True)`` recebe dois argumentos, o objeto a ser escrito e se ele tem que ser escapado (definido como ``True`` por padrão). Se ``obj`` tem um ``.xml()`` método, ele é chamado e o resultado escrito para o corpo da resposta (o argumento ``escape`` é ignorado). Caso contrário, ele usa ``\_\_str\_\_`` o método do objeto para serializar-lo e, se o argumento fuga é ``True``, lhe escapa. Todos os built-in helper objetos (``H1`` no exemplo) são objetos que sabem como serializar-se através do ``.xml()`` método.

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 Sintaxe básica

The py4web template language supports all Python control structures. Here we provide some examples of each of them. They can be nested according to usual programming practice. 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 `` Para ... in ``

Em templates você pode fazer um loop sobre qualquer objeto iterável:

```
[[items = ['a', 'b', 'c']]]

[[for item in items:]][[=item]][[pass]]

```

que produz:

```

a
b
c

```

Aqui `` items `` é qualquer objeto iterável como uma lista Python, Python tupla, ou linhas objeto, ou qualquer objeto que é implementado como um iterador. Os elementos apresentados são primeiro serializado e escapou.

### 9.1.2 `` While ``

Você pode criar um loop usando a palavra-chave, enquanto:

```
[[k = 3]]

[[while k > 0:]][[=k]] [[k = k - 1]][[pass]]

```

que produz:

```

3
2
1

```

### 9.1.3 `` If ... elif ... else ``

Você pode usar cláusulas condicionais:

```
[[
import random
k = random.randint(0, 100)
]]
<h2>
[[=k]]
[[if k % 2:]]is odd[[else:]]is even[[pass]]
</h2>
```

que produz:

```
<h2>
45 is odd
</h2>
```

Uma vez que é óbvio que ``else`` encerra a primeira ``if`` bloco, não há necessidade de um ``declaração pass``, e usando um seria incorreto. No entanto, você deve fechar explicitamente a opção ``bloco else`` com um ``pass``.

Lembre-se que, em Python “else if” está escrito ``elif`` como no exemplo a seguir:

```
[[
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>
```

Produz:

```
<h2>
64 is divisible by 4
</h2>
```

### 9.1.4 ``Tentar ... exceto ... 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:]]

[[pass]]
```

Ela irá produzir o seguinte resultado:

```
Hello division by zero


```

Este exemplo ilustra que todas as saídas gerado antes de ocorrer uma exceção é processado (incluindo a saída que precedeu a exceção) no interior do bloco de teste. “Olá” é escrito porque precede a exceção.

### 9.1.5 ``Def ... return``

O linguagem de template py4web permite ao desenvolvedor definir e implementar funções que podem retornar qualquer objeto Python ou uma cadeia de texto / html. Aqui, consideramos dois exemplos:

```
[[def itemize1(link): return LI(A(link, _href="http://" + link))]]

[[=itemize1('www.google.com')]]

```



produz o seguinte resultado:

```

www.google.com

```

A função ``itemize1`` devolve um objecto auxiliar que é inserido no local em que a função é chamada.

Considere agora o seguinte código:

```
[[def itemize2(link):]]
[[link]]
[[return]]

[[itemize2('www.google.com')]]

```

Ela produz exactamente o mesmo resultado como acima. Neste caso, a função ``itemize2`` representa um pedaço de HTML que vai substituir a tag py4web onde a função é chamada. Observe que não existe '=' na frente da chamada para ``itemize2``, já que a função não retornar o texto, mas escreve-lo diretamente para a resposta.

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 original template 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,AAABAAEAAQEAABAAIAAAwAAAAFgAAACgAAAABAAAAAgAAAAEAIAAAAAABAAAAAA
 <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 -->

 py4web
<script>document.write(window.location.href.split('/')[3]);</script>

 <!-- 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 -->

 [[if globals().get('user')]]

 <a class="navbar-link is-primary"
 [[=globals().get('user', {}).get('email')]]

 Edit Profile
 Change
Password
 Logout

 [[else]]

 Login

 Sign up
 Log in

 </header>
 </body>
 </html>
```

```
 [[pass]]

</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">
 <!-- content 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.

---

## Helpers YATL

---

### 10.1 Helpers overview

Consider the following code in a template:

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

ele é processado como:

```
<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

Em vez de um conjunto de argumentos sem nome, um helper também pode ter uma única lista ou tupla como seu conjunto de componentes usando a notação ``*`` e pode levar um único dicionário como seu conjunto de atributos usando o `**``, por exemplo:

```
[[
 contents = ['this', 'is', 'a', 'test']
 attributes = {'_id': '123', '_class': 'myclass'}
 =DIV(*contents, **attributes)
]]
```

(Produce 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")]]
```

é prestado:

```
<div class="myclass"><i>hello <world></i></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).

Componentes de helpers podem ser referenciados através de sua posição, e helpers agir como listas com relação aos seus componentes:

```
>>> a = DIV(SPAN('a', 'b'), 'c')
>>> print(a)
<div>abc</div>
>>> del a[1]
>>> a.append(STRONG('x'))
>>> a[0][0] = 'y'
>>> print(a)
<div>ybx</div>
```

Atributos de helpers pode ser referenciado pelo nome, e helpers agir como dicionários com relação aos seus atributos:

```
>>> a = DIV(SPAN('a', 'b'), 'c')
>>> a['_class'] = 's'
>>> a[0]['_class'] = 't'
>>> print(a)
<div class="s">abc</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.

Note que atributos auxiliares são passados como argumentos para o auxiliar. Em alguns casos, no entanto, nomes de atributos incluem caracteres especiais que não são permitidos em identificadores Python (por exemplo, hífens) e, portanto, não podem ser usados como nomes de argumentos de palavra-chave. Por exemplo:

```
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>
```

Você também pode criar dinamicamente tags especiais:

```
>>> 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.

O texto neste exemplo é escapado:

```
>>> print (DIV("hello"))
<div>hello</div>
```

usando ``XML`` você pode impedir escapar:

```
>>> print (DIV(XML("hello")))
<div>hello</div>
```

Às vezes você quer renderizar HTML armazenado em uma variável, mas o HTML pode conter tags inseguras como 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))
<script>alert("unsafe!")</script>
```

Os ``construtores XML``, por padrão, considere o conteúdo de algumas tags e alguns de seus atributos de segurança. Você pode substituir os padrões usando os opcionais ``permitted\_tags`` e ``allowed\_attributes`` argumentos. Aqui estão os valores padrão dos argumentos opcionais do ``helper XML``.

```
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``

Este assistente é usado para construir ligações.

```
>>> print (A('<click>', XML('me'),
 _href='http://www.py4web.com'))
<click>me
```

### 10.2.3 ``BODY``

Este assistente faz com que o corpo de uma página.

```
>>> print (BODY('<hello>', XML('world'), _bgcolor='red'))
<body bgcolor="red"><hello>world</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 link, and here is some bold text.
```

### 10.2.5 ``Div``

This is the content division element.

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

### 10.2.6 ``EM``

Insiste no seu conteúdo.

```
>>> print(EM('<hello>', XML('world'), _class='test', _id=0))
<em id="0" class="test"><hello>world
```

### 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('world'), _class='test', _id=0))
<h1 id="0" class="test"><hello>world</h1>
```

### 10.2.9 ``HEAD``

Para marcar a cabeça de uma página HTML.

```
>>> print(HEAD(TITLE('<hello>', XML('world'))))
<head><title><hello>world</title></head>
```

### 10.2.10 ``HTML``

For tagging an HTML page.

```
>>> print(HTML(BODY('<hello>', XML('world'))))
<html><body><hello>world</body></html>
```

### 10.2.11 ``I``

Este assistente torna o seu conteúdo em itálico.

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

### 10.2.12 ``IMG``

It can be used to embed images into HTML.

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

```

Aqui é uma combinação de helpers A, IMG, e URL para a inclusão de uma imagem estática com um link:

```
>>> print(A(IMG(_src=URL('static', 'logo.png'), _alt="My Logo"),
... _href=URL('default', 'index')))

```

### 10.2.13 ``INPUT``

Cria um ``<input ... />`` tag. Uma tag de entrada não pode conter outras tags, e é fechada por ``/>`` em vez de ``>``. A tag de entrada tem um atributo opcional ``\_type`` que pode ser definido como “texto” (o padrão), “enviar”, “caixa”, ou “rádio”.

```
>>> 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
```

e similarmente para caixas de seleção:

```
>>> 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``

Ele é usado para criar uma tag rótulo para um campo de entrada.

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

### 10.2.15 ``LI``

Faz um item da lista e deve estar contido em um ``UL`` ou ``tag OL``.

```
>>> print(LI('<hello>', XML('world'), _class='test', _id=0))
<li id="0" class="test"><hello>world
```

### 10.2.16 ``OL``

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

```
>>> print(OL(LI('<hello>'), LI(XML('world')), _class='test',
_id=0))
<ol class="test" id="0"><hello>world
```

### 10.2.17 `` OPTION ``

This should only be used as argument of a SELECT.

```
>>> print(OPTION('<hello>', XML('world'), _value='a'))
<option value="a"><hello>world</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 ``

Isto é para marcar um parágrafo.

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

### 10.2.19 `` PRE ``

Gera um `` `<pre> ... </pre>` `` tag para exibir texto pré-formatado. O `` `CODE` `` auxiliar é geralmente preferível para listagens de código.

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

Semelhante a `` `div` `` mas utilizado para marcação em linha (em vez de bloco) conteúdo.

```
>>> print(SPAN('<hello>', XML('world'), _class='test', _id=0))
<hello>world
```

### 10.2.23 `` STYLE ``

Semelhante ao `script`, mas usadas para incluir ou código do link CSS. Aqui, o CSS está incluído:

```
>>> print(STYLE(XML('body {color: white;}'))
<style>body {color: white}</style>
```

e aqui ela está ligada:

```
>>> print(STYLE(_src='style.css'))
<style src="style.css"></style>
```

### 10.2.24 ``TABLE``, ``TR``, ``TD``

Estas tags (juntamente com o opcional ``THEAD`` e ``helpers TBODY``) são utilizados para tabelas de construção HTML.

```
>>> 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.

É fácil converter uma matriz de Python em uma tabela HTML usando ``\*`` notação argumentos de função do Python, que mapeia os elementos da lista para os argumentos da função posicionais.

Aqui, vamos fazê-lo linha por linha:

```
>>> 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>
```

Aqui nós fazer todas as linhas de uma só vez:

```
>>> 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``

Isto é usado para linhas tag contidos no corpo de mesa, em oposição a linhas de cabeçalho ou de rodapé. É opcional.

```
>>> print(TBODY(TR(TD('<hello>')), _class='test', _id=0))
<tbody id="0" class="test"><tr><td><hello></td></tr></tbody>
```

### 10.2.26 ``TEXTAREA``

Este assistente faz uma `<textarea> ... </textarea>` tag ``.

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

### 10.2.27 ``TH``

Este é utilizado em vez de ``TD`` em cabeçalhos de tabela.

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

### 10.2.28 ``THEAD``

Isto é usado para linhas de cabeçalho da tabela tag.

```
>>> print(THEAD(TR(TH('<hello>')), _class='test', _id=0))
<thead id="0" class="test"><tr><th><hello></th></tr></thead>
```

### 10.2.29 ``TITLE``

Isto é usado para marcar o título de uma página em um cabeçalho HTML.

```
>>> print(TITLE('<hello>', XML('world')))
<title><hello>world</title>
```

### 10.2.30 ``TT``

Etiquetas de texto como máquina de escrever texto (monoespaçada).

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

### 10.2.31 ``UL``

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

```
>>> print(UL(LI('<hello>'), LI(XML('world')), _class='test',
_id=0))
<ul class="test" id="0"><hello>world
```

### 10.2.32 ``URL``

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

## 10.3 Helpers personalizados

### 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')]
```

gera o seguinte XML:

```
<name c="d">ab</name>
```

Argumentos "a", "b" e "d" são automaticamente escapou; usar o ``helper XML`` para suprimir esse comportamento. Usando ``TAG`` você pode gerar HTML / XML marcas já não fornecidos pela API. As etiquetas podem ser aninhados, e são serializados com ``str()`` Uma sintaxe é equivalente.:

```
[TAG['name']('a', 'b', _c='d')]
```

Tags com auto-fechamento podem ser geradas com o helper TAG. O nome da tag deve terminar com um "/".

```
[TAG['link/'](_href='http://py4web.com')]
```

gera o seguinte 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`` é usado para representações de construção HTML de objetos compostos, incluindo listas, tuplas e dicionários:

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

``BEAUTIFY`` retorna um objeto serializado XML-like to XML, com uma representação de vista agradável de seu argumento construtor. Neste caso, a representação XML:

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

retribuirá como:

```
<table><tbody>
<tr><th>a</th><td>helloworld</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:

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

that returns all the components matching supplied arguments.

A very simple query can be a tag name:

```
>>> 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>z3<div>y</div></div></div>
>>> for c in a.find('span'): c[0]='z'
>>> print(a)
<div><div>z3<div>z</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': '$'})), P('world',
_class='this is a test'))
>>> for e in a.find('div a#1-1, p.is'): print(e)
hello
<p class="this is a test">world</p>
>>> for e in a.find('#1-1'): print(e)
hello
>>> a.find('a[u:v=$]')[0].xml()
'hello'
>>> 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>
```

Se `` substituir = None``, os elementos correspondentes serão completamente removidas.

```
>>> 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>x<div>z</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>hello<div>hellohello</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>x<div>helloz</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:

```
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**:

```
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

```
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]]`

---

## Internacionalização

---

### 11.1 Pluralizar

Pluralizar é uma biblioteca Python para a Internacionalização (i18n) e Pluralização (p10n).

The library assumes a folder (for example “translations”) that contains files like:

```
it.json
it-IT.json
fr.json
fr-FR.json
(etc)
```

Cada arquivo tem a seguinte estrutura, por exemplo para o italiano (it.json):

```
{"dog": {"0": "no cane", "1": "un cane", "2": "{n} cani", "10": "tantissimi cani"}}
```

As chaves de nível superior são as expressões a ser traduzido e o valor associado / dicionário mapeia um número para uma tradução. Diferentes traduções correspondem a diferentes formas de plural da expressão,

Aqui está outro exemplo para a palavra “cama” em checo

```
{"bed": {"0": "no postel", "1": "postel", "2": "postele", "5": "postelí"}}
```

Para traduzir e pluralizar de “cachorro” string um simplesmente deforma a corda na operadora T da seguinte forma:

```
>>> 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
```

A cadeia pode conter vários espaços reservados, mas o {n} espaço reservado é especial porque

a variável chamada “n” é usado para determinar a pluralização pelo melhor jogo (tecla dict max <= n).

T (...) os objetos podem ser adicionados em conjunto com os outros e com a corda, como cordas regulares.

T.select (s) pode analisar uma string s seguinte HTTP aceito formato de idioma.

## 11.2 Atualizar os arquivos de tradução

Encontrar todas as cordas envoltas em T (...) em .py, .html e arquivos .js:

```
matches = T.find_matches('path/to/app/folder')
```

Adicione entradas recém-descobertas em todos os idiomas suportados

```
T.update_languages(matches)
```

Add a new supported language (for example German, “de”)

```
T.languages['de'] = {}
```

Certifique-se de todos os idiomas contêm as mesmas expressões de origem

```
known_expressions = set()
for language in T.languages.values():
 for expression in language:
 known_expressions.add(expression)
T.update_languages(known_expressions)
```

Finalmente salvar as alterações:

```
T.save('translations')
```

---

## Formulários

---

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

O ``construtor Form`` aceita os seguintes argumentos:

```
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,
):

```

Onde:

- `table`: a DAL table or a list of fields
- ``Record``: um registro DAL ou ID de registro
- ``Readonly``: Defina como true para fazer um formulário readonly
- ``Deletable``: definida para Falso ao apagamento disallow de registro
- - **formstyle: a function that renders the form using helpers.**  
Can be `FormStyleDefault` (default), `FormStyleBulma`, `FormStyleBootstrap4`, or `FormStyleBootstrap5`.
- ``Dbio``: definida para Falso para evitar quaisquer gravações DB
- ``Keep\_values``: se definido como verdadeiro, ele lembra os valores do formulário previamente submetidas
- ``Form\_name``: o nome opcional desta forma
- ``Hidden``: um dicionário de campos ocultos que é adicionado à forma
- `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`:

```
in controllers.py
from py4web import 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](http://127.0.0.1:8000/form_minimal) - you'll get the Form page:

Name

Enter a value

Color

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`:

```
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>

[[for row in rows:]]
[[=row.id]]: [[=row.name]] has color [[=row.color]]
[[pass]]

```

Reload py4web and visit [http://127.0.0.1:8000/create\\_form](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

Image

Upload:  No file chosen

## 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>

[[for row in rows:]]
[[=row.id]]: [[=row.name]] has color [[=row.color]]

[[pass]]

```

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:

```
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 an existing style. Let’s have a quick look, improving again our Superhero example:

```
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 Validação de formulário

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 first 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 validator 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,  $0 \leq \text{value} \leq 100$  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,**

$0 \leq \text{value} < 100$  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)
```

### IS\_IN\_SET and Tagging

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 from 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')
```

Onde:

- 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 useful 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:

```
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.

```
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:

```
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.

Aqui está um exemplo:

```
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:

- nome do usuário
- o email
- senha
- primeiro nome
- último nome
- sso\_id (usado para single sign on, ver mais adiante)
- action\_token (usado para verificar e-mail, bloquear usuários e outras tarefas, também ver mais adiante).

A `` `auth.enable()` `` passo cria e expõe os seguintes APIs RESTful:

- {Nomeaplic} / auth / api / registo (POST)
- {Nomeaplic} / auth / api / Login (POST)
- {Nomeaplic} / auth / api / request\_reset\_password (POST)
- {Nomeaplic} / auth / api / reset\_password (POST)
- {Appname} / auth / api / verify\_email (GET, POST)
- {Nomeaplic} / auth / api / Sair (GET, POST) (+)
- {Nomeaplic} / auth / api / perfil (GET, POST) (+)
- {Nomeaplic} / auth / api / change\_password (POST) (+)
- {Nomeaplic} / auth / api / change\_email (POST) (+)

Os que estão marcados com um (+) requerem um usuário conectado.

### 13.1.1 Interface de autenticação

Você pode criar sua própria interface do usuário da web para usuários de login usando as APIs acima, mas py4web fornece um como exemplo, implementada nos seguintes arquivos:

- \_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:

```

 [[if globals().get('user')]]

 [[=globals().get('user', {}).get('email')]]

 Edit Profile
 [[if 'change_password' in
globals().get('actions', {}).get('allowed_actions', {}):]]
 Change Password
 [[pass]]
 Logout

 [[else:]]

 Login

 Sign up
 Log in

 [[pass]]

```

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.

```
@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:

```
@action('index')
@action.uses(auth.user)
def index():
 user = auth.get_user()
 return 'hello {first_name}'.format(**user)
```

Aqui `` @ action.uses (auth.user) `` diz py4web que essa ação requer um usuário conectado e deve redirecionar para login se nenhum usuário está logado.

### 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.

```
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:

```
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:

```
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 Plugins de Autenticação

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).

O `` <auth /> `` componentes irá se adaptar automaticamente para formulários de login de exibição, conforme exigido pelos plugins instalados.

In the `_scaffold/settings.py` and `_scaffold/common.py` files you can see the default settings for

the supported plugins.

## PAM

Configurando PAM é o mais fácil:

```
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).

O ``auth.register\_plugin (...)`` must **not** vir antes do ``auth.enable ()``, uma vez que não faz sentido para expor APIs antes de plugins desejados são montados.

---

**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'))
```

O ID de cliente e segredo do cliente deve ser fornecido pelo 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'))
```

O ID de cliente e segredo do cliente deve ser fornecido pelo 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 to support authentication. The distinction 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 sessions. 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 is no need for any kind of additional API token which would only diminish 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 `@action.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 database 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 verifying 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 exactly 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 Etiquetas e permissões

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`:

The screenshot shows a web interface titled "Databases in \_scaffold". It displays two database tables:

- db.auth\_user**: Fields include id, username, email, password, first\_name, last\_name, sso\_id, action\_token, last\_password\_change, and past\_passwords\_hash.
- db.auth\_user\_tag\_groups**: Fields include id, path, and record\_id.

Então você pode adicionar uma ou mais marcas de registros da tabela, bem como remover existente 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):

The screenshot shows a web interface titled "App: \_scaffold Database: db Table: auth\_user\_tag\_groups". It includes a search filter "filter (example id > 1)" and buttons for "Search" and "Create". Below the filter, it indicates "(2 items found)".

Id	Path	Record Id
[1]	/manager/	[1]
[2]	/teacher/	[1]

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'
```

Aqui o desenvolvedor consulta o banco de dados para todos os registros que têm a tag desejada (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'
```

Deixamos para você como um exercício para criar um dispositivo elétrico ``has\_membership`` para permitir a seguinte sintaxe:

```
@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 ( "professor /-ensino médio / física") ``
- `` Groups.find ( "professor /-colegial") ``
- `` Groups.find ( "professor") ``

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')
```

E você pode verificar se há uma permissão de utilizador através de uma junção explícita:

```
@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'
```

Aviso aqui `` permissions.find (permissão) `` gera uma consulta para todos os grupos com a permissão e que ainda filtro desses grupos para aqueles do utilizador actual é membro da. Contamos eles e se encontrarmos qualquer, então o usuário tem a permissão.

### 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.

```
@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"))
```

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

- CRUD completa com Confirmação de exclusão
- Clique cabeças de coluna para classificar - clique novamente para DESC
- Controle de paginação
- Construído em Search (pode usar search\_queries OU search\_form)
- Botões de ação - com ou sem texto
- Pré e Pós Ação (adicionar seus próprios botões para cada linha)
- Datas de grid em formato local
- Formatação padrão por tipo de utilizador mais substituições

---

**Hint** There is an excellent grid tutorial made by Jim Steil on [https://github.com/jpsteil/grid\\_tutorial](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.

```
in grid/__init__.py
import os
from py4web import action, Field, DAL
from py4web.utils.grid import *
from py4web.utils.form import *
from yat1.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()

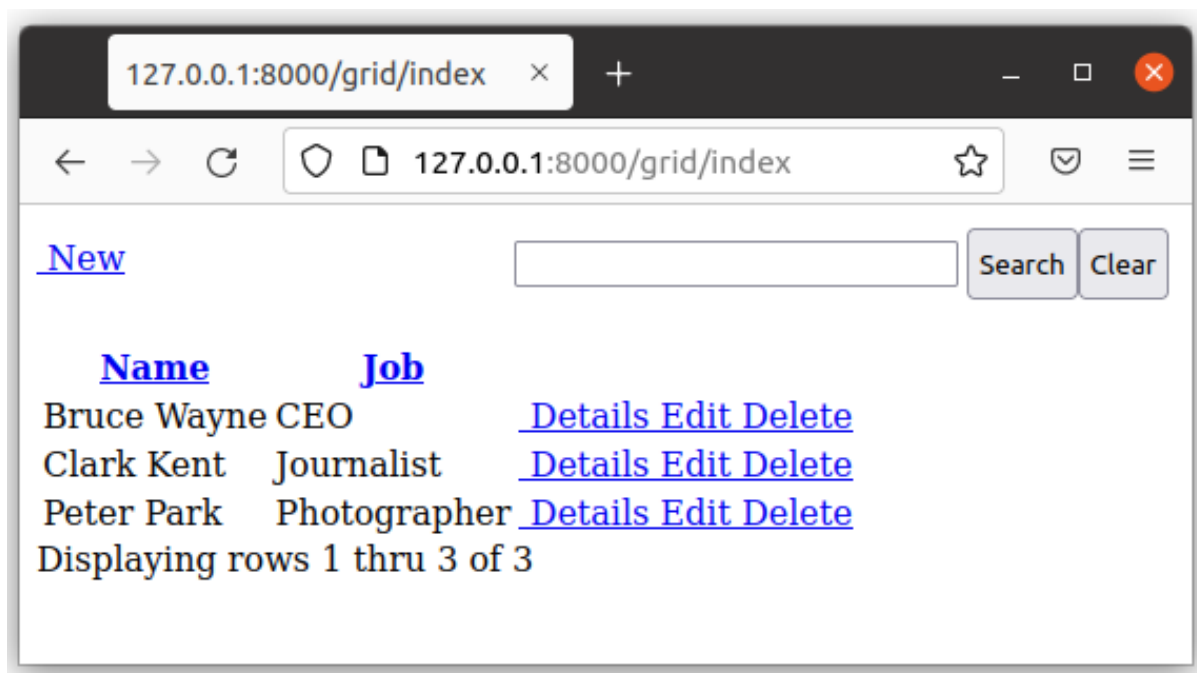
@action('index', method=['POST', 'GET'])
@action.uses('grid.html', db)
def index():
 grid = Grid(
 formstyle=FormStyleDefault, # FormStyleDefault, FormStyleBulma,
 FormStyleBootstrap4, or FormStyleBootstrap5
 grid_class_style=GridClassStyle, # GridClassStyle or
 GridClassStyleBulma or GridClassStyleBootstrap5
 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:

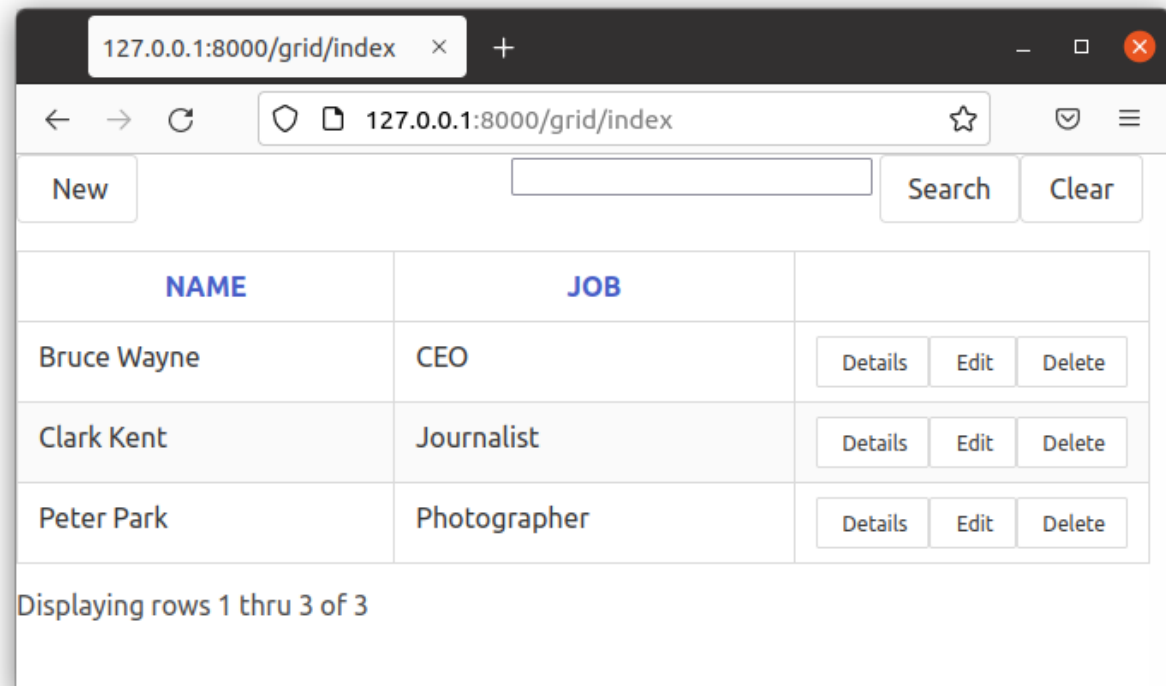
```
<!DOCTYPE html>
<html>
 <head>
 <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.9.3/css/bulma.min.css">
```

```

</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](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=IconStyleFontawesome,
 T=lambda text: text,
):

```

- caminho: a rota do pedido
- query: consulta pydal a ser processado
- 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: mostrar o campo ID de registo na página de lista - default = false
- orderby: Campo orderby pydal ou lista de campos
- esquerda: se juntando outras tabelas, especifique a expressão esquerda pydal aqui
- títulos: lista de títulos a ser utilizado para página da lista - se não for fornecido o uso do rótulo do campo
- criar: URL para redirecionar para a criação de registros - definido como verdadeiro para gerar automaticamente o URL - definido como falso para não exibir o botão
- 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: lista de instâncias action\_button para incluir antes de os botões de ação padrão
- post\_action\_buttons: lista de instâncias action\_button para incluir após os botões de ação padrão
- auto\_process: Boolean - se ou não a grid deve ser processado imediatamente. Se False, desenvolvedor deve chamar grid.process () uma vez todos os parâmetros são configurados

- `rows_per_page`: número de linhas a serem exibidos por página. padrão 15
- `include_action_button_text`: boolean dizendo a grid se deseja ou não de texto em botões de ação dentro de sua grid
- `search_button_text`: texto a ser exibido no botão enviar em seu formulário de pesquisa
- `formstyle`: py4web Form formstyle usado para estilo seu formulário ao construir automaticamente formulários CRUD
- `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:

- Fornecer uma lista `search_queries`
- Construa a sua própria forma de pesquisa personalizada

Se você fornecer uma lista `search_queries` à grid, ele irá:

- 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
- recolher valores de filtro e filtrar a grid

No entanto, se isso não lhe dá flexibilidade suficiente, você pode fornecer o seu próprio formulário de busca e lidar com toda a filtragem (construção das consultas) por si mesmo.

### 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.

Além disso, você pode fornecer uma URL separada para a criação / detalhes / editáveis / parâmetros elimináveis para ignorar as páginas CRUD gerados automaticamente e lidar com as páginas de detalhes do mesmo.

## 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 yat1.helpers import A

columns = [
```



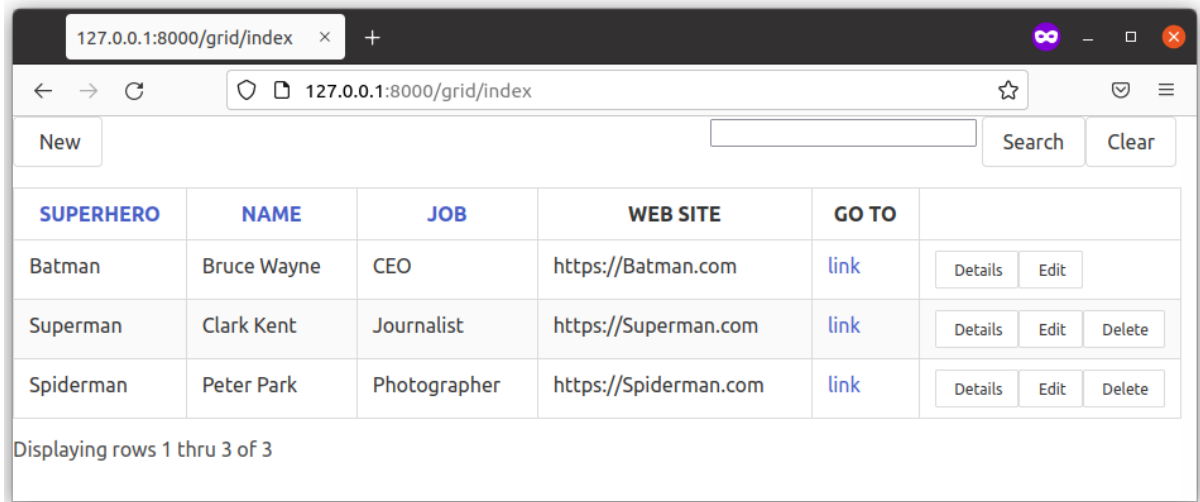
```

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 Usando templates

Use o seguinte para tornar a sua grid ou formas CRUD em seus templates.

Mostrar a grid ou um formulário CRUD

```
[[=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

Você pode fornecer suas próprias formstyle ou grid classes e estilo ao grid.

- formstyle é o mesmo que um formstyle Forma, usadas para as formas estilo CRUD.
- grid\_class\_style é uma classe que fornece as classes e / ou estilos utilizados para certas porções da grelha.
- 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 Ação personalizada Botões

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 Botão Classe Ação Amostra

```
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: a página para navegar até quando o botão é clicado
- texto: texto para exibição no botão
- icon: the icon corresponding to your icon-style to display before the text, for example «fa-calendar» for IconStyleFontawesome.
- additional\_classes: uma lista separada por espaços de aulas para incluir no elemento botão
- 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
- mensagem: mensagem de confirmação para exibição se a classe 'confirmação' é adicionado a classes adicionais
- append\_id: Se for verdade, adicionar id\_field\_name = id\_value à querystring url para o botão
- 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=IconStyleFontawesome.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 Os campos de referência

Ao exibir campos em uma tabela PyDAL, às vezes você deseja exibir um campo mais descritivo do que um valor de chave estrangeira. Há um par de maneiras de lidar com isso com a grid py4web.

`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 ''),

```

Isto irá exibir o nome da empresa na grid em vez do ID empresa

A queda de usar este método é que classificação e filtragem são baseados no campo da empresa na tabela de empregado e não o nome da empresa

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))

```

Este método permite classificar e filtrar, mas não permite que você para combinar campos a serem exibidos em conjunto, como o método filter\_out faria

Você precisa determinar qual método é melhor para o seu caso de uso compreender as grids diferentes no mesmo aplicativo pode precisar de se comportar de forma diferente.

## 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:

```

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).



---

## De web2py para py4web

---

Este capítulo é dedicado a ajudar os usuários para portar aplicativos web2py antigos para 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 overwritten 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

```
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

```
@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

```
def counter():
 session.counter = (session.counter or 0) + 1
 return str(session.counter)
```

→ py4web

```
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): }}
{{= k }}
{{ pass }}
</div>
```

→ py4web

```
[[extend 'layout.html']]
<div>
[[for k in range(1):]]
[[= k]]
[[pass]]
</div>
```

## 15.1.8 “Form and flash” example

web2py

```
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

```
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

```
<div class="flash">[[globals().get('flash', '')]]</div>
```

or using the more sophisticated

```
<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

```
def index():
 grid = SQLFORM.grid(db.thing, editable=True)
 return locals()
```

→ py4web

```
@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

```
file_path = os.path.join(request.folder, 'file.csv')
```

→ py4web

```
from .settings import APP_FOLDER
file_path = os.path.join(APP_FOLDER, 'file.csv')
```

### 15.1.11 “auth” example

web2py

```
auth = Auth()
auth.define_tables()
```

```
@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**

```
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:

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

and

```
@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:

```
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 using 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

```
def my_task():
 try:
 # do something
 db.commit()
 except Exception:
 db.rollback()
```

## 16.2 Sending messages using a background task

As an 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:

```
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:

```
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:

```
class MySendGridSender:
 def __init__(self, from_addr):
 self.from_addr = from_addr
 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 is 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

```

[[for sh in superheros:]]
 <a hx-get="[[=URL('htmx_form/%s' % sh.id)]]"
 hx-target="#htmx-form-demo">[[sh.name]]
[[pass]]

```

#### templates/htmx\_form.html

```
[[=form]]
```

We now have a functional maintenance app to update our superheroes. 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.

```
import json
from functools import reduce

from yat1 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)
```

```

 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 `HtmxAutocompleteWidget` 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("received", response); }
var failure = response => { console.log("received", 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="browsers-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.

- : Ref: `genindex`
- : Ref: `modindex`
- : Ref: `search`





