

# **py4web Documentation**

*Release 1.20201127.1*

**© 2020, BSDv3 License**

dezembro 08, 2020



<b>1</b>	<b>O que é py4web?</b>	<b>1</b>
<b>2</b>	<b>Ajuda, recursos e dicas</b>	<b>3</b>
2.1	Recursos . . . . .	3
2.2	Dicas e sugestões . . . . .	4
2.3	Como contribuir . . . . .	4
<b>3</b>	<b>Instalação e colocação em funcionamento</b>	<b>7</b>
3.1	Plataformas e pré-requisitos suportados. . . . .	7
3.2	Procedimentos de configuração. . . . .	7
3.3	Melhoramento . . . . .	9
3.4	Primeira corrida. . . . .	9
3.5	Opções de linha de comando . . . . .	11
3.6	Implantação na nuvem . . . . .	14
<b>4</b>	<b>O Dashboard</b>	<b>17</b>
4.1	A página Web principal . . . . .	17
4.2	Sessão no Dashboard . . . . .	18
<b>5</b>	<b>Criando seu primeiro aplicativo</b>	<b>27</b>
5.1	Do princípio . . . . .	27
5.2	Páginas estáticas. . . . .	27
5.3	Páginas web dinâmicas . . . . .	28
5.4	De _scaffold . . . . .	31
5.5	App Watchdog. . . . .	32
<b>6</b>	<b>Fixures</b>	<b>35</b>
6.1	Importante sobre Fixures. . . . .	35
6.2	Templates Py4web. . . . .	36
6.3	Sessões . . . . .	36
6.4	Tradutor . . . . .	38
6.5	O fixture flash . . . . .	39
6.6	O fixture DAL . . . . .	40
6.7	Advertências sobre Fixures. . . . .	41
6.8	Fixtures personalizados. . . . .	41
6.9	Auth e Auth.user . . . . .	42
6.10	Caching e Memoize . . . . .	43

6.11	Decoradores de conveniência . . . . .	43
<b>7</b>	<b>A camada de abstracção de banco de dados (DAL)</b>	<b>45</b>
7.1	Dependências . . . . .	45
7.2	A DAL: Um rápido passeio. . . . .	46
7.3	Usando o DAL “stand-alone” . . . . .	46
7.4	Construtor DAL . . . . .	47
7.5	Construtor Table. . . . .	52
7.6	Construtor Field . . . . .	55
7.7	Migrações . . . . .	60
7.8	`` Insert`` . . . . .	62
7.9	`` `` commit`` e rollback`` . . . . .	62
7.10	Raw SQL . . . . .	63
7.11	`` Drop`` . . . . .	64
7.12	Índices . . . . .	64
7.13	Bancos de dados legados e tabelas com chave . . . . .	65
7.14	Transação distribuída . . . . .	65
7.15	Mais sobre envios . . . . .	66
7.16	`` Query``, `` Set``, `` Rows`` . . . . .	67
7.17	`` Comando SELECT`` . . . . .	68
7.18	Outros métodos . . . . .	79
7.19	Campos computados . . . . .	80
7.20	Campos virtuais . . . . .	81
7.21	Um para muitos relação . . . . .	84
7.22	Muitos para muitos . . . . .	87
7.23	Marcação de registros. . . . .	88
7.24	`` Lista: <type> `` e `` contains`` . . . . .	89
7.25	Outros operadores . . . . .	90
7.26	Gerando SQL puro . . . . .	94
7.27	Exportar e importar dados . . . . .	95
7.28	Selects com cache . . . . .	99
7.29	A auto-referência e aliases . . . . .	100
7.30	Características avançadas . . . . .	101
7.31	Pegadinhas . . . . .	112
<b>8</b>	<b>A RESTAPI</b>	<b>115</b>
8.1	RestAPI GET . . . . .	116
<b>9</b>	<b>Linguagem de template YATL</b>	<b>133</b>
9.1	Sintaxe básica. . . . .	134
<b>10</b>	<b>Helpers YATL</b>	<b>139</b>
10.1	`` XML`` . . . . .	141
10.2	Built-in helpers. . . . .	142
10.3	Helpers personalizados . . . . .	149
10.4	`` BEAUTIFY`` . . . . .	151
10.5	Server-side * DOM * e análise. . . . .	151
10.6	Layout da página. . . . .	154
10.7	Funções em vista . . . . .	160
10.8	Blocos em vista. . . . .	160
<b>11</b>	<b>Internacionalização</b>	<b>163</b>
11.1	Pluralizar . . . . .	163

11.2	Atualizar os arquivos de tradução . . . . .	164
<b>12</b>	<b>Formulários</b>	<b>165</b>
12.1	Exemplo. . . . .	165
12.2	Validação de formulário. . . . .	166
<b>13</b>	<b>Autenticação e controle de acesso</b>	<b>167</b>
13.1	Interface de autenticação . . . . .	168
13.2	Usando o Auth . . . . .	168
13.3	Plugins de Autenticação. . . . .	169
13.4	Etiquetas e permissões . . . . .	170
<b>14</b>	<b>Rede</b>	<b>173</b>
14.1	Características principais . . . . .	173
14.2	Exemplo básico. . . . .	173
14.3	Assinatura . . . . .	174
14.4	Searching / Filtering. . . . .	175
14.5	CRUD . . . . .	175
14.6	Usando templates . . . . .	176
14.7	Personalizando Estilo . . . . .	176
14.8	Ação personalizada Botões . . . . .	180
14.9	Botão Classe Ação Amostra. . . . .	180
14.10	Os campos de referência . . . . .	180
<b>15</b>	<b>De web2py para py4web</b>	<b>183</b>



---

# O que é py4web?

---

PY4WEB é um framework web para desenvolvimento rápido de aplicações web banco de dados orientado eficientes. É uma evolução do quadro web2py popular, mas muito mais rápido e mais lisa. Seu design interno tem sido muito simplificada em comparação com web2py.

PY4WEB pode ser visto como um concorrente de outros frameworks como Django ou Flask, e pode de fato servir ao mesmo propósito. No entanto, objetivos PY4WEB para fornecer um conjunto de recursos maior fora da caixa e reduzir o tempo de desenvolvimento de novos aplicativos.

De uma perspectiva histórica nossa história começa em 2007, quando web2py foi lançado pela primeira vez. web2py foi projetado para fornecer uma solução all-inclusive para desenvolvimento web: um zip arquivo contendo o interpretador Python, o quadro, um baseado na web IDE, e uma coleção de pacotes de batalha-testadas que funcionam bem juntos. De muitas maneiras web2py tem sido extremamente bem sucedido. Web2py conseguiu proporcionar uma baixa barreira de entrada para novos desenvolvedores, uma plataforma de desenvolvimento muito seguro e permanece para trás compatíveis até hoje.

Web2py sempre sofreu de um problema: seu design monolítico. Os desenvolvedores do Python mais experientes não entender como usar seus componentes fora do quadro e como usar componentes de terceiros no quadro. Este foi por uma boa razão, uma vez que não se importam muito sobre eles. Pensamos em web2py como uma ferramenta perfeita que não tem que ser quebrado em pedaços, porque isso iria comprometer a sua segurança. Descobriu-se que estávamos errados, e jogar bem com os outros é importante. Assim, desde 2015, trabalhou em três frentes:

- 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 é mais do que uma reembalagem desses módulos. É uma completa remodelação. Ele usa alguns dos módulos web2py, mas não todos eles. Em alguns casos, ele usa outras e melhores módulos. Alguns funcionalidade foi removido e foi adicionado alguns. Nós tentamos preservar a maior parte da sintaxe e recursos que os usuários web2py experientes amado. Aqui está uma lista mais explícito:

- PY4WEB, ao contrário web2py, requer Python 3.
- PY4WEB, ao contrário web2py, pode ser instalado usando pip e suas dependências são gerenciados usando requirements.txt.
- Aplicativos PY4WEB são módulos regulares Python. Isto é muito diferente para web2py. Em particular, abandonou o importador de costume, e contamos agora exclusivamente no mecanismo regular de importação Python.
- PY4WEB, como web2py, podem servir múltiplas aplicações concorrentemente, enquanto as aplicações são submódulos do módulo de aplicações.

- PY4WEB, ao contrário web2py, é baseado em bottlepy e em usos particulares do objeto do pedido Garrafa e o mecanismo de roteamento de Garrafa.
- PY4WEB, ao contrário web2py, não cria um novo ambiente em cada solicitação. Ele introduz o conceito de luminárias para explicitamente declarar que objetos precisam ser quando uma nova solicitação HTTP é processado inicializado-re. Isso torna muito mais rápido.
- PY4WEB, tem um novo objeto session que, como a do web2py, fornece segurança forte e criptografia dos dados da sessão, mas as sessões não são mais armazenadas no sistema de arquivos - o que criou problemas de desempenho. Ele fornece sessões de cookies, em Redis, no memcache, ou no banco de dados. Também limita os dados da sessão a objetos que são JSON serializado.
- 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, como web2py, é baseado em pydal mas usa alguns novos recursos de 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 vem com um objeto Form que é semelhante ao SQLFORM do web2py mas é muito mais simples e mais rápido. A sintaxe é a mesma. Este foi fornecido, a fim de apps existentes portuários ajudam os usuários; mas PY4WEB incentiva usando formas API baseada sobre postbacks.
- 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 vem com alguns utilitários como “tags”, por exemplo, que permite adicionar tags pesquisáveis a qualquer banco de dados tabela. Ele pode ser usado, por exemplo, para usuários de tag com grupos e usuários de pesquisa por grupos e aplicar permissões com base na associação.
- PY4WEB vem com 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.



---

## 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

Este manual é o Manual de Referência para py4web. Está disponível on-line em [https://py4web.com/\\_documentation/static/index.html](https://py4web.com/_documentation/static/index.html), onde você também encontrará o PDF e versão e-book, em vários idiomas. Ele foi originalmente escrito com o formato MarkMin (semelhante ao Markdown) e exibidas em HTML com uma aplicação py4web personalizado. Em 2020, decidi converter suas fontes para o formato RST que é mais adequado para documentação técnica. Usando Esfinge eo estilo ReadTheDocs agora somos capazes de alcançar resultados de alta qualidade.

#### 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 O bate-papo no IRC

Nós também usamos para conversar em algum momento no IRC (Internet Relay Chat, que é um texto de estilo antigo única chat). Você pode se juntar a nós livremente no <https://webchat.freenode.net/#py4web>. De vez em quando nós também usá-lo para hospedar um bate-papo pública agendada, onde você pode escrever e ler perguntas ao vivo para os desenvolvedores. Transcrições deles são, então, disponível na lista de discussão.

#### 2.1.4 Tutoriais e vídeo

Existem alguns tutoriais e vídeos, se você gosta deles. Procurá-los na YouTube <[https://www.youtube.com/results?search\\_query=py4web](https://www.youtube.com/results?search_query=py4web)> \_\_. Há também disponível um livre bom curso por Luca de Alfaro <<https://sites.google.com/a/ucsc.edu/luca/classes/cmsp-183-hypermedia-and-the-web/cse-183-spring-2020>> \_\_ na UC Santa Cruz.

## 2.1.5 As fontes no GitHub

Py4web é Open Source, com uma licença BSD v3. Ele está hospedado no GitHub em <https://github.com/web2py/py4web>, onde você pode ler e estudar todos os seus detalhes internos.

## 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

Nos capítulos seguintes, você vai começar a codificar em seu computador. Sugerimos que você configurar um moderno local de trabalho python se você pretende fazê-lo de forma eficiente e segura. Mesmo para a execução de exemplos simples e experimentar um pouco, sugerimos usar um **Ambiente de Desenvolvimento Integrado** (IDE). Isso fará com que a sua experiência de programação muito melhor, permitindo verificação de sintaxe, linting e depuração visual. Hoje em dia, existem dois principais escolhas livres e multiplataforma: Microsoft Visual Código Estúdio aka *VScode* <<https://code.visualstudio.com/>> \_\_ e JetBrains *PyCharm* <<https://www.jetbrains.com/pycharm/>> \_\_.

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

- usar ambientes virtuais (também chamado **virtualenv**, veja *aqui* <<https://docs.python.org/3.7/tutorial/venv.html>> \_\_ para uma introdução). Em um ambiente de trabalho complexo isso vai evitar a ser confuso com outros programas Python e módulos
- usar um Concurrent Versions System (CVS **\*\***). Manter o controle de mudanças do seu programa é muito valioso - juntamente com backups que vai salvar sua vida computação! Git e GitHub são os padrões atuais.

### 2.2.3 Depuração py4web com VScode

Você precisa editar o arquivo de configuração vscode launch.json adicionando a variável ``«GEvent»: True``. Além disso, no mesmo arquivo você pode adicionar ``«args»: [ «run», «apps»]``, a fim de executar py4web.py diretamente.

### 2.2.4 Depuração py4web com PyCharm

Em PyCharm, ative Configurações | Build, Execução, Implantação | Python Debugger | GEvent compatível.

## 2.3 Como contribuir

Precisamos da ajuda de todos: apoiar os nossos esforços! Você pode apenas participe no grupo Google

tentando responder a outras das perguntas, enviar bugs usando ou criar pedidos puxe o repositório GitHub.

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.



---

## Instalação e colocação em funcionamento

---

### 3.1 Plataformas e pré-requisitos suportados

PY4WEB funciona muito bem no Windows, MacOS e Linux. Seu único pré-requisito é Python 3.6+, que deve ser instalado com antecedência (exceto se você usar os binários).

### 3.2 Procedimentos de configuração

Existem quatro formas alternativas de correr py4web, com diferentes níveis de dificuldade e flexibilidade. Vamos olhar os prós e contras.

#### 3.2.1 Instalando a partir de binários

Esta não é uma instalação real, porque você acabou de copiar um monte de arquivos em seu sistema sem modificá-lo de qualquer maneira. Daí esta é a solução mais simples, especialmente para iniciantes ou alunos, porque ele não requer Python pré-instalado em seu sistema, nem direitos administrativos. Por outro lado, é experimental, poderia conter uma liberação py4web de idade e é muito difícil para adicionar outras funcionalidades a ele.

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-start set_password
py4web-start run apps
```

Com este tipo de instalação, lembre-se de usar sempre `py4web-start ** **` em vez de `'py4web'` ou `'py4web.py'` na seguinte documentação.

#### 3.2.2 Dica: use um ambiente virtual (virtualenv)

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>> \_\_ 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!

Ativá-lo antes de usar qualquer um dos seguintes *\* real \** procedimentos de instalação é altamente

recomendado.

### 3.2.3 Instalando a partir de pip

Usando pip `**` é o procedimento de instalação padrão para py4web. A partir da linha de comando

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

mas `**` não `**` digite o `-user` opção com virtualenv ou uma instalação padrão do Windows que já por usuário é.

Além disso, se `python3` não funcionar, tente com o simples `python` comando em vez.

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
```

Se o py4web comando não é aceito, isso significa que ele não está no caminho do sistema. No Windows, um arquivo py4web.exe especial (apontando para py4web.py) será criado por `pip` no caminho do sistema, mas não se você digitar o `-user` opção por engano.

### 3.2.4 Instalação de fonte (globalmente)

Esta é a maneira tradicional para a instalação de um programa, mas ele só funciona em Linux e MacOS. Todos os requisitos será instalado no caminho do sistema, juntamente com links para o programa py4web.py na pasta local

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

Note também que ao instalar desta forma o conteúdo de py4web / ativo pasta está faltando no começo, mas ele é criado manualmente mais tarde com o comando `'make activos'`.

### 3.2.5 Instalando a partir de fonte (localmente)

Desta forma todos os requisitos será instalado ou atualizado no caminho do sistema, mas a própria py4web só serão copiados em uma pasta local. Isto é especialmente útil se você já tem uma instalação py4web de trabalho, mas você quer testar uma diferente one. From the command line, go to a given working folder and then run

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

Uma vez instalado, você deve sempre começar a partir daí com

`** Para Linux / MacOS **`

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

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

Se você tiver instalado py4web tanto global como localmente, observe a **\*\* / \*\***; ele força o prazo de py4web da pasta local e não o instalado globalmente.

#### Para Windows

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

No Windows, os programas na pasta local são sempre executados antes de os do caminho (portanto, você não precisa do **\*\* / \*\***). Mas a execução de arquivos .py diretamente não é habitual e você vai precisar de um comando explícito python3 / python.

## 3.3 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 apps
```

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

Se você instalou py4web de qualquer outra maneira, você deve atualizá-lo manualmente. Primeiro você tem que fazer um backup de qualquer trabalho py4web pessoal que você fez, em seguida, elimine a pasta de instalação de idade e re-instalar o quadro novamente.

## 3.4 Primeira corrida

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

```
# py4web run apps
```

```
PY4WEB
Is still experimental...

Py4web: 1.2020112.1 on Python 3.8.6 (default, Sep 25 2020, 09:36:53)
[GCC 10.2.0]

Dashboard is at: http://127.0.0.1:8000/_dashboard
[X] loaded _default
[X] loaded _dashboard
[X] loaded examples
[X] loaded _documentation
[X] loaded myfeed
[X] loaded _scaffold
[X] loaded todo
[X] loaded _minimal
Bottle v0.12.18 server starting up (using TornadoServer())...
Listening on http://127.0.0.1:8000/
Hit Ctrl-C to quit.
```

Geralmente ``apps`` é o nome da pasta onde você guarda todos os seus aplicativos, e pode ser definido explicitamente usando o comando ``run``. Se essa pasta não existir, ele será criado. PY4WEB espera encontrar pelo menos dois aplicativos nesta pasta: \*\* Painel \*\* (\_dashboard) e \*\* Padrão \*\* (\_default). Se não encontrá-los, ele instala-os.

\*\* 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 nomes reais com ``\_`` para evitar conflitos com aplicativos criados por você.

---

Uma vez py4web está sendo executado você pode acessar um aplicativo específico nas seguintes URLs:

```
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.

Para todas as aplicações de arrastamento `` / index`` é opcional.

## 3.5 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.

### 3.5.1 Opção `` comando call``

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

    Call a function inside apps_folder

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

### 3.5.2 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:
  -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.

### 3.5.3 Opção `` comando run``

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

  Run all the applications on apps_folder

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

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

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

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

Se você quiser py4web para recarregar automaticamente uma aplicação sobre quaisquer alterações nos arquivos desse aplicativo, você pode:

- para recarregamento imediato (sync-modo): ``py4web prazo -watch = sync``
- para recarregar em qualquer primeira solicitação de entrada para a aplicação foi alterada (de modo lento): ``py4web prazo -watch = lazy``

### 3.5.4 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

```
py4webt 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: *****
```

### 3.5.5 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.

### 3.5.6 Opção `` comando shell``

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

    Open a python shell with apps_folder added to the path

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

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
```

### 3.5.7 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.
```

Com o \* -a \* opção você vai ter a versão de todos os módulos python disponíveis também.

## 3.6 Implantação na nuvem

### 3.6.1 Implantação em gcloud (aka Google App Engine)

Entrada no console do gcloud (<https://console.cloud.google.com/>) e criar um novo projeto. Você vai obter um ID de projeto que se parece com “{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\_\_ vazio \_\_. 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``.

### 3.6.2 Implantação em PythonAnywhere.com

Assista ao vídeo: [https://youtu.be/Wxjl\\_vkLAEY](https://youtu.be/Wxjl_vkLAEY) e siga o tutorial detalhado sobre <https://github.com/tomcam/py4webcasts/blob/master/docs/how-install-source-pythonanywhere.md>. O script `bottle_app.py` é em ``py4web / deployment\_tools / pythonanywhere.com / bottle\_app.py``



---

## **O Dashboard**

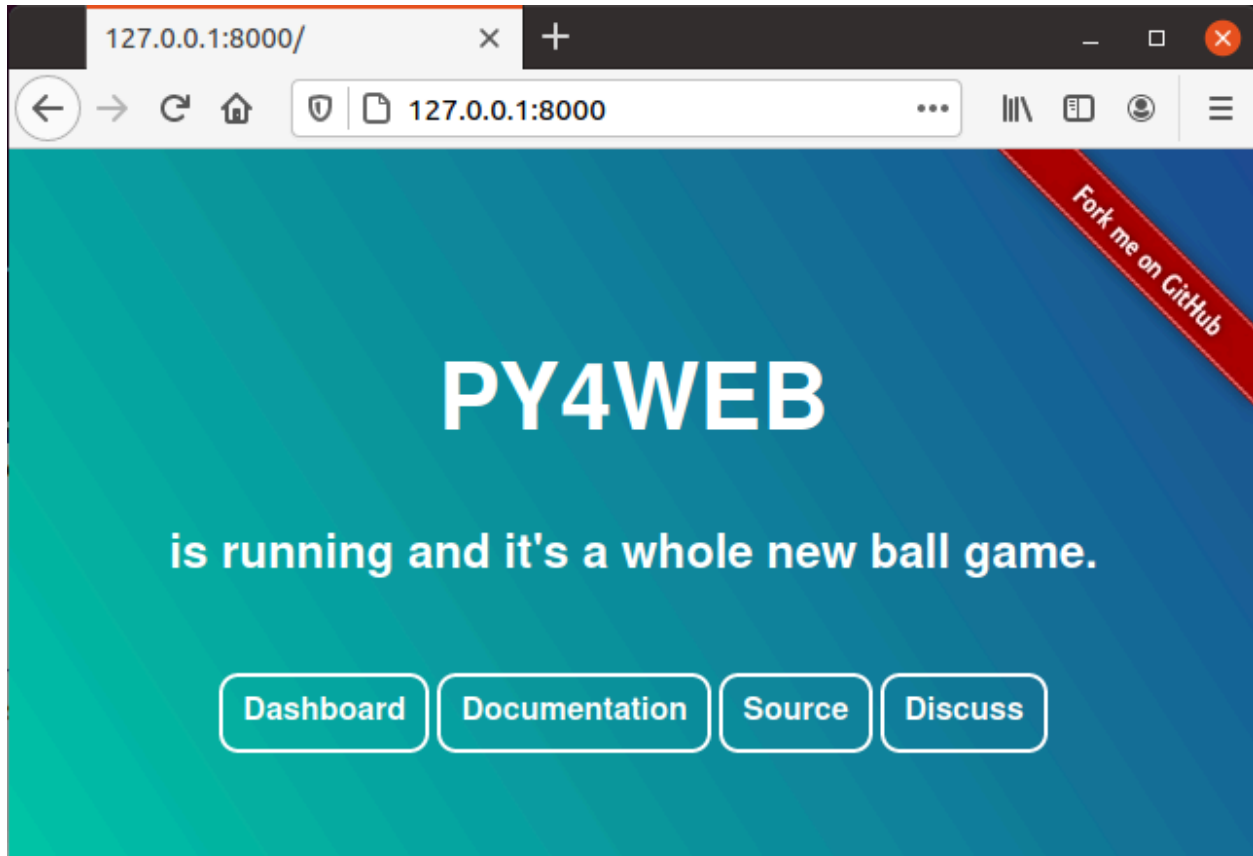
---

O Dashboard é o padrão IDE baseado na web; você certamente irá usá-lo extensivamente para gerir as aplicações e verificar bancos de dados. Olhando para a sua interface é uma boa maneira de começar a explorar py4web e seus componentes.

### **4.1 A página Web principal**

Quando você executar o programa py4web padrão, ele começa um servidor web com uma página web de audição principal em <http://127.0.0.1:8000> (o que significa que ele está escutando na porta TCP 8000 no seu PC local, usando o protocolo http).

Você pode conectar-se a esta página apenas a partir de seu PC local, usando um navegador web como o Firefox ou o Google Chrome:



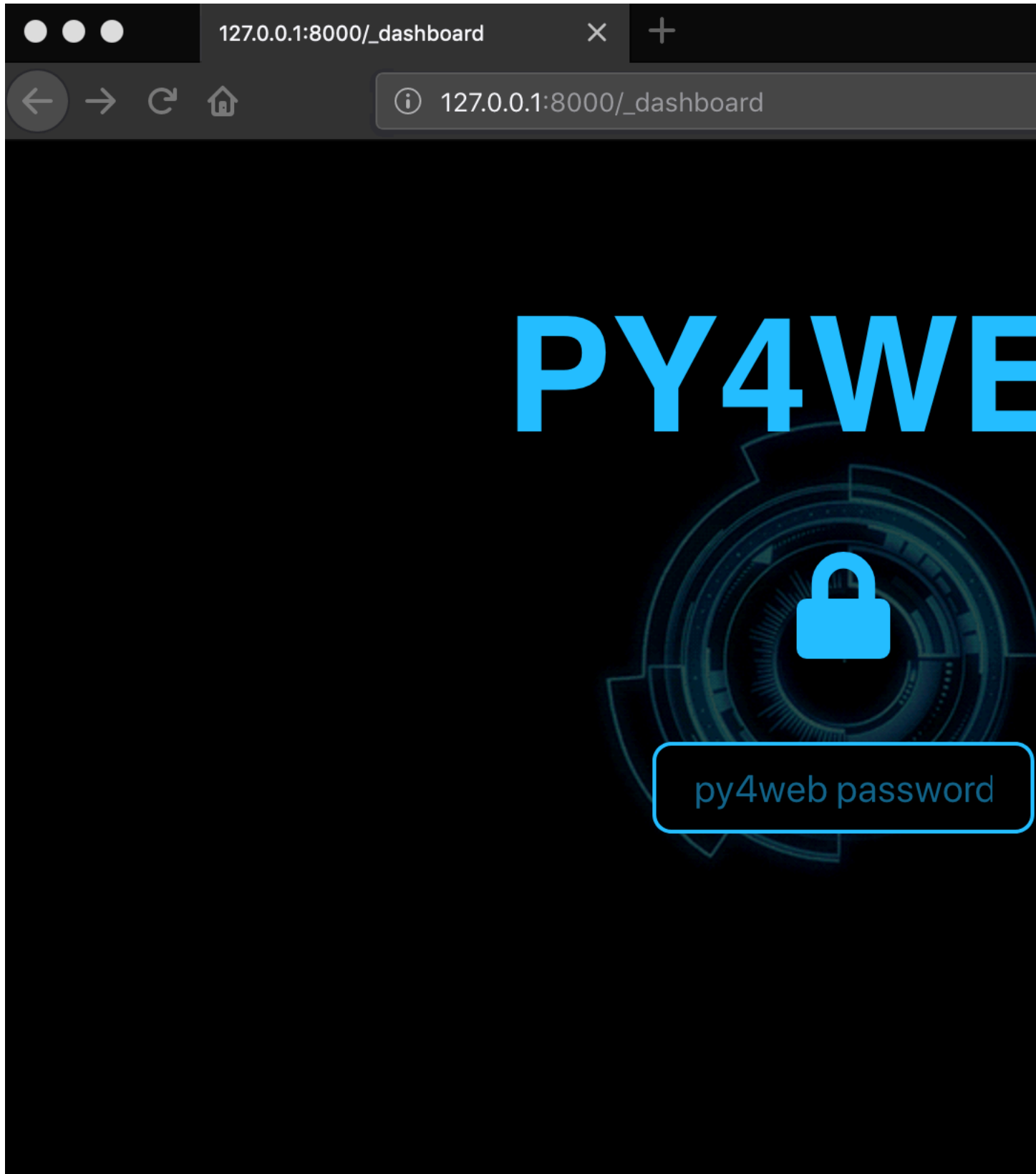
Os botões são:

- Dashboard ([http://127.0.0.1:8000/\\_dashboard](http://127.0.0.1:8000/_dashboard)), que iremos descrever neste capítulo
- Documentação ([http://127.0.0.1:8000/\\_documentation?version=1.20201127.1](http://127.0.0.1:8000/_documentation?version=1.20201127.1)), para navegar na cópia local deste Manual
- Fonte (<https://github.com/web2py/py4web>), apontando para o repositório GitHub
- Discutir (<https://groups.google.com/forum/#!forum/py4web>), apontando para o grupo mail do Google

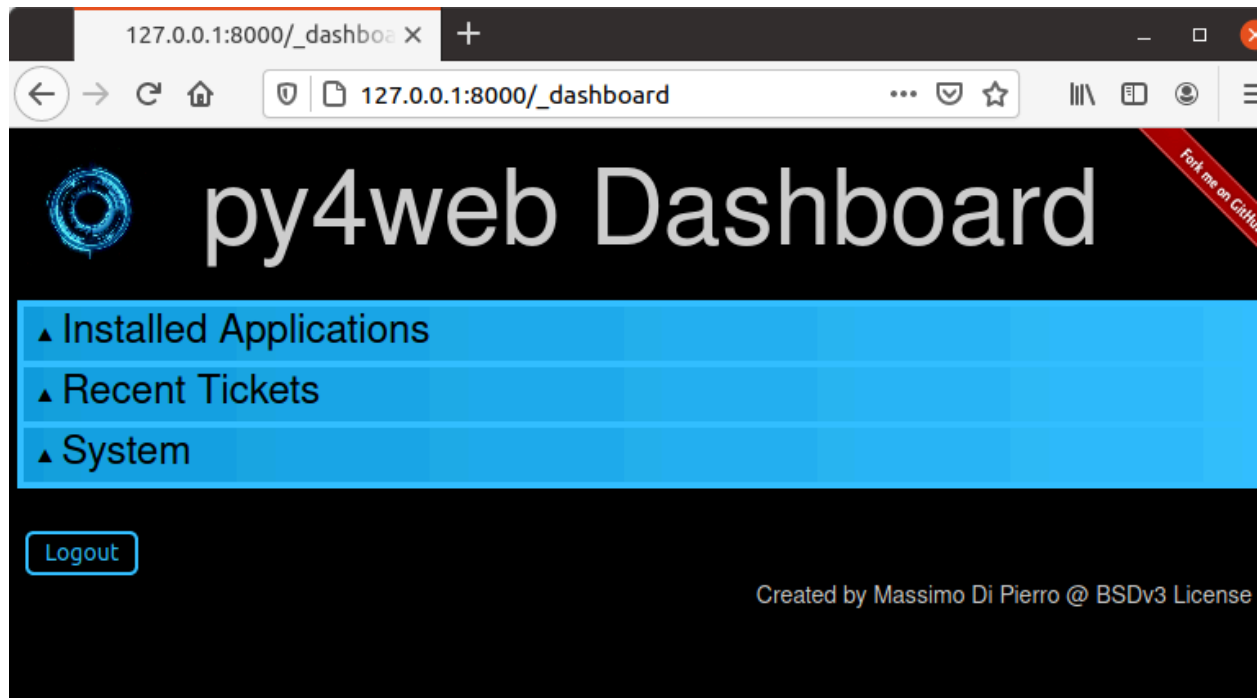
## 4.2 Sessão no Dashboard

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.

▼ Installed Applications

\_dashboard    \_default    \_documentation  
 \_scaffold    examples    home  
 static    superheroes    templates

▼ Routes for todo

Rule	Method	Filename	Action
/todo	GET	todo/___init__.py	index
/todo/api	GET	todo/___init__.py	todo
/todo/api	POST	todo/___init__.py	todo
/todo/api/<id:int>	DELETE	todo/___init__.py	todo
/todo/index	GET	todo/___init__.py	index
/todo/uuid	GET	todo/___init__.py	uuid

▼ Files in todo

A aba “Arquivos” permite que você navegue a pasta que contém o aplicativo selecionado e editar qual-

quer arquivo que inclui o aplicativo. Se você editar um arquivo que você deve clicar em “Recarregar Apps” sob a guia “Aplicativos instalados” para que a alteração tenha efeito (exceto se você usar \* relógio \* com o: ref: *comando de execução option*). Se um aplicativo não for carregado, o botão correspondente é exibido em vermelho. Clique na imagem para ver o erro correspondente.

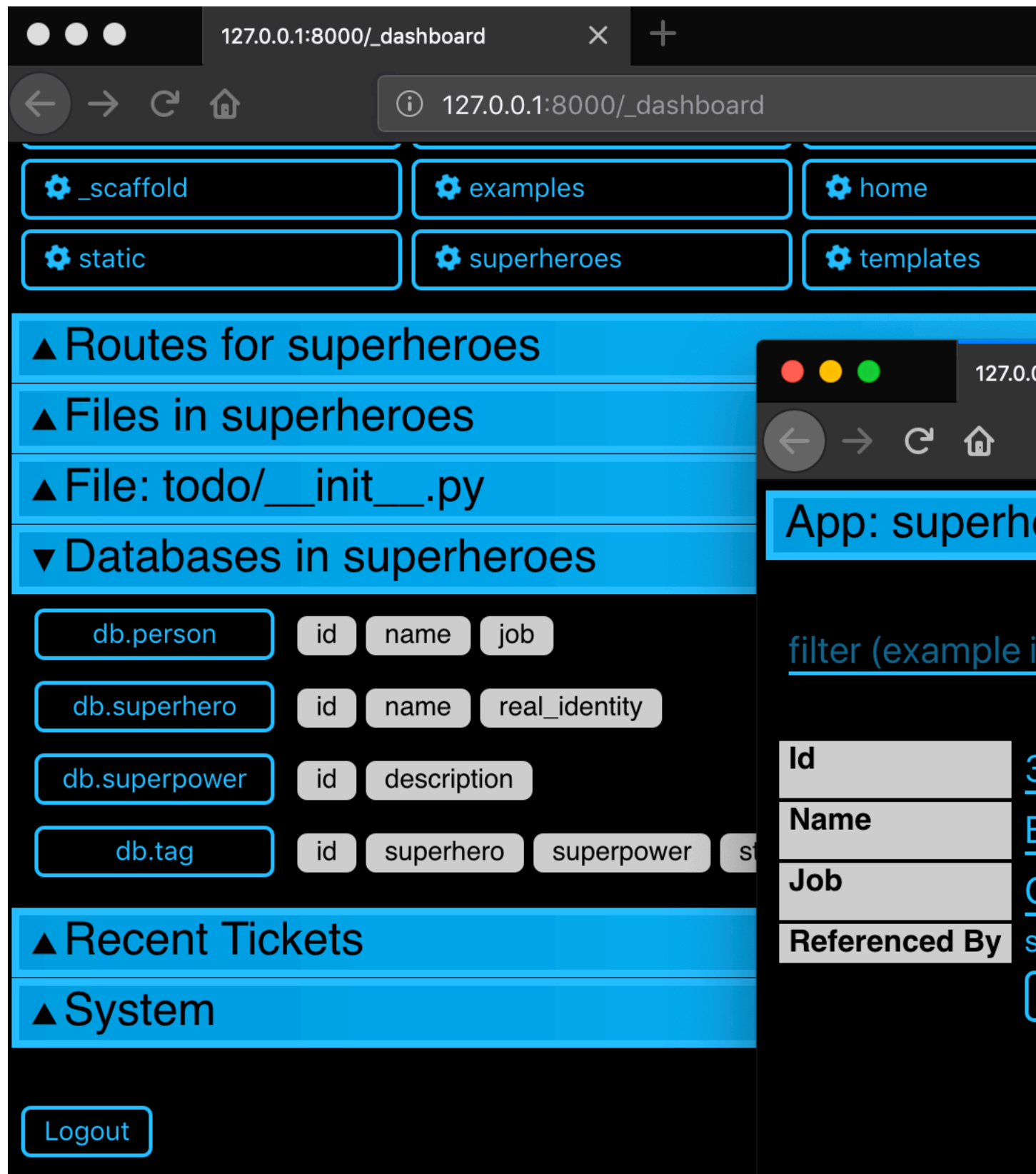
```

1 import os
2 from py4web import action, request, DAL, Field, Session, Cache, use
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
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 a
14 @action.uses('index.html') # we use the template index.htm
15 @action.uses(session) # action needs a session object
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 us
26 @action.uses(db) # all before starting a db conn
27 def todo():
28     return dict(items=db(db.todo).select(orderby=~db.todo.id).as_li
29

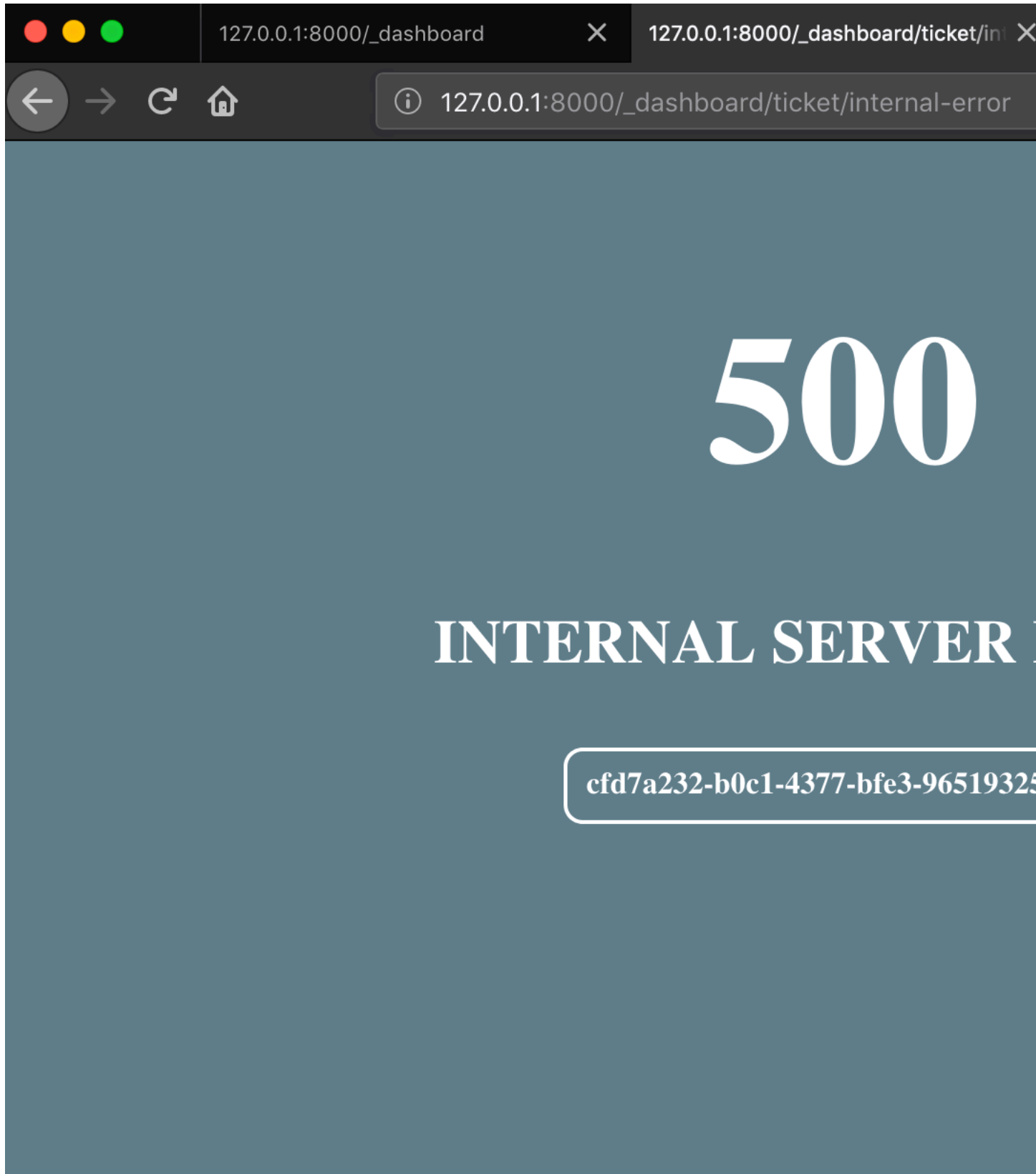
```

O painel expõe o db de todas as aplicações que utilizam RESTAPI pydal. Ele também fornece uma inter-

face web para realizar operações de busca e CRUD.



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.

<b>id:</b>	5																												
<b>uuid:</b>	625267d8-f4b1-467e-bce8-2e43807ba925																												
<b>app_name:</b>	examples																												
<b>method:</b>	GET																												
<b>path:</b>	/examples/oops																												
<b>timestamp:</b>	2019-09-03 00:12:48																												
<b>client_ip:</b>	127.0.0.1																												
<b>error:</b>	division by zero																												
<b>snapshot:</b>	<table border="1"> <tr> <td><b>timestamp:</b></td> <td>2019-09-03T00:12:48.110126</td> </tr> <tr> <td><b>python_version:</b></td> <td>3.7.0b4 (v3.7.0b4:eb96c37699, May 2 2018 [Clang 6.0 (clang-600.0.57)])</td> </tr> <tr> <td><b>platform_info:</b></td> <td> <table border="1"> <tr> <td><b>machine:</b></td> <td>x86_64</td> </tr> <tr> <td><b>node:</b></td> <td>me</td> </tr> <tr> <td><b>platform:</b></td> <td>Darwin-18.6.0-x86_64</td> </tr> <tr> <td><b>processor:</b></td> <td>i386</td> </tr> <tr> <td><b>python_branch:</b></td> <td>v3.7.0b4</td> </tr> <tr> <td><b>python_build:</b></td> <td> <ul style="list-style-type: none"> <li>v3.7.0b4:eb96c37699</li> <li>May 2 2018</li> </ul> </td> </tr> <tr> <td><b>python_compiler:</b></td> <td>Clang 6.0 (clang-600.0.57)</td> </tr> <tr> <td><b>python_implementation:</b></td> <td>CPython</td> </tr> <tr> <td><b>python_revision:</b></td> <td>eb96c37699</td> </tr> <tr> <td><b>python_version:</b></td> <td>3.7.0b4</td> </tr> <tr> <td><b>python_version_tuple:</b></td> <td>3, 7, 0</td> </tr> </table> </td> </tr> </table>	<b>timestamp:</b>	2019-09-03T00:12:48.110126	<b>python_version:</b>	3.7.0b4 (v3.7.0b4:eb96c37699, May 2 2018 [Clang 6.0 (clang-600.0.57)])	<b>platform_info:</b>	<table border="1"> <tr> <td><b>machine:</b></td> <td>x86_64</td> </tr> <tr> <td><b>node:</b></td> <td>me</td> </tr> <tr> <td><b>platform:</b></td> <td>Darwin-18.6.0-x86_64</td> </tr> <tr> <td><b>processor:</b></td> <td>i386</td> </tr> <tr> <td><b>python_branch:</b></td> <td>v3.7.0b4</td> </tr> <tr> <td><b>python_build:</b></td> <td> <ul style="list-style-type: none"> <li>v3.7.0b4:eb96c37699</li> <li>May 2 2018</li> </ul> </td> </tr> <tr> <td><b>python_compiler:</b></td> <td>Clang 6.0 (clang-600.0.57)</td> </tr> <tr> <td><b>python_implementation:</b></td> <td>CPython</td> </tr> <tr> <td><b>python_revision:</b></td> <td>eb96c37699</td> </tr> <tr> <td><b>python_version:</b></td> <td>3.7.0b4</td> </tr> <tr> <td><b>python_version_tuple:</b></td> <td>3, 7, 0</td> </tr> </table>	<b>machine:</b>	x86_64	<b>node:</b>	me	<b>platform:</b>	Darwin-18.6.0-x86_64	<b>processor:</b>	i386	<b>python_branch:</b>	v3.7.0b4	<b>python_build:</b>	<ul style="list-style-type: none"> <li>v3.7.0b4:eb96c37699</li> <li>May 2 2018</li> </ul>	<b>python_compiler:</b>	Clang 6.0 (clang-600.0.57)	<b>python_implementation:</b>	CPython	<b>python_revision:</b>	eb96c37699	<b>python_version:</b>	3.7.0b4	<b>python_version_tuple:</b>	3, 7, 0
<b>timestamp:</b>	2019-09-03T00:12:48.110126																												
<b>python_version:</b>	3.7.0b4 (v3.7.0b4:eb96c37699, May 2 2018 [Clang 6.0 (clang-600.0.57)])																												
<b>platform_info:</b>	<table border="1"> <tr> <td><b>machine:</b></td> <td>x86_64</td> </tr> <tr> <td><b>node:</b></td> <td>me</td> </tr> <tr> <td><b>platform:</b></td> <td>Darwin-18.6.0-x86_64</td> </tr> <tr> <td><b>processor:</b></td> <td>i386</td> </tr> <tr> <td><b>python_branch:</b></td> <td>v3.7.0b4</td> </tr> <tr> <td><b>python_build:</b></td> <td> <ul style="list-style-type: none"> <li>v3.7.0b4:eb96c37699</li> <li>May 2 2018</li> </ul> </td> </tr> <tr> <td><b>python_compiler:</b></td> <td>Clang 6.0 (clang-600.0.57)</td> </tr> <tr> <td><b>python_implementation:</b></td> <td>CPython</td> </tr> <tr> <td><b>python_revision:</b></td> <td>eb96c37699</td> </tr> <tr> <td><b>python_version:</b></td> <td>3.7.0b4</td> </tr> <tr> <td><b>python_version_tuple:</b></td> <td>3, 7, 0</td> </tr> </table>	<b>machine:</b>	x86_64	<b>node:</b>	me	<b>platform:</b>	Darwin-18.6.0-x86_64	<b>processor:</b>	i386	<b>python_branch:</b>	v3.7.0b4	<b>python_build:</b>	<ul style="list-style-type: none"> <li>v3.7.0b4:eb96c37699</li> <li>May 2 2018</li> </ul>	<b>python_compiler:</b>	Clang 6.0 (clang-600.0.57)	<b>python_implementation:</b>	CPython	<b>python_revision:</b>	eb96c37699	<b>python_version:</b>	3.7.0b4	<b>python_version_tuple:</b>	3, 7, 0						
<b>machine:</b>	x86_64																												
<b>node:</b>	me																												
<b>platform:</b>	Darwin-18.6.0-x86_64																												
<b>processor:</b>	i386																												
<b>python_branch:</b>	v3.7.0b4																												
<b>python_build:</b>	<ul style="list-style-type: none"> <li>v3.7.0b4:eb96c37699</li> <li>May 2 2018</li> </ul>																												
<b>python_compiler:</b>	Clang 6.0 (clang-600.0.57)																												
<b>python_implementation:</b>	CPython																												
<b>python_revision:</b>	eb96c37699																												
<b>python_version:</b>	3.7.0b4																												
<b>python_version_tuple:</b>	3, 7, 0																												



---

## Criando seu primeiro aplicativo

---

### 5.1 Do princípio

Aplicativos podem ser criados usando o painel ou diretamente a partir do sistema de arquivos. Aqui, nós estamos indo para fazê-lo manualmente, como o Dashboard é descrito em seu próprio capítulo.

Tenha em mente que um aplicativo é um módulo Python; portanto, precisa apenas de uma pasta e um `__init__.py` nessa pasta.:

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

**Tip** para Windows, você deve usar barras invertidas (ou seja, `"\"`) em vez de barras. Além disso, um vazio `*__init__.py*` arquivo não é estritamente necessária desde Python 3.3, mas será útil mais tarde.

Se você agora reiniciar py4web ou pressione o botão “Atualizar Apps” no Dashboard, py4web vai encontrar este módulo, importá-lo e reconhecê-lo como um aplicativo, simplesmente por causa de sua localização. Um aplicativo não é obrigado a fazer nada. Poderia ser apenas um recipiente para arquivos estáticos ou código arbitrário que outros aplicativos pode querer importação e de acesso. No entanto, normalmente a maioria dos aplicativos são projetados para expor páginas estáticas ou dinâmicas.

### 5.2 Páginas estáticas

Para expor páginas estáticas você simplesmente precisa para criar um `static` subpasta 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.

Importante: py4web internamente usa a garrafa `static_file` <<https://bottlepy.org/docs/dev/tutorial.html#static-file>>

*ic-files*> \_\_ método para servir arquivos estáticos, o que significa que suporta streaming, conteúdo parcial, pedidos de intervalo, e If-Modified-Since. Isso tudo é feito automaticamente com base nos cabeçalhos de solicitação HTTP.

## 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()
```

Py4web Restart ou pressione o botão “Atualizar Apps” botão do painel, e esta página será acessível em

```
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

ações py4web deve retornar uma string ou um dicionário. Se eles retornar um dicionário que você deve dizer py4web o que fazer com ele. Por padrão py4web serializará-lo em JSON. Por exemplo editar `` \_\_init \_\_. Py`` novamente e adicionar

```
@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
- ``\*\*\*``

E estes são possíveis filtros (somente ``:`` tem uma 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]``` permite especificar uma expressão regular personalizado no campo de configuração. O valor combinado não é modificado.

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

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
```

Observe que o objeto de solicitação é um *pedido objeto Garrafa* <<https://bottlepy.org/docs/dev/api.html#the-request-object>> \_.

### 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 precisarmos de ambos. As dependências são tratados automaticamente.

## 5.4 De \_scaffold

Na maioria das vezes, você não quer começar a escrever código a partir do zero. Você também quer seguir algumas convenções sãs descritas aqui, como não colocar todo seu código em ``\_\_init\_\_.py``. PY4WEB fornece um aplicativo de Andaimes (\_scaffold), onde os arquivos são organizados de forma adequada e muitos objetos úteis são pré-definidos.

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``:

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

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

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

@action('welcome', method='GET')
@action.uses('generic.html', session, db, T, auth.user)
def index():
```

```
user = auth.get_user()
message = T('Hello {first_name}'.format(**user))
return dict(message=message, user=user)
```

Observe o seguinte:

- `` Request``, response``, `` abort`` são definidos por garrafa
- `` Redirect`` e URL`` são semelhantes aos seus homólogos web2py
- ajudantes ( `` A``, `` div``, `` SPAN``, `` IMG``, etc.) deve ser importado a partir `` yatl.helpers``. Eles trabalham muito bem como em 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 App Watchdog

Py4web facilita a configuração de um servidor de desenvolvimento que recarrega automaticamente um aplicativo quando sua fonte Python arquivos mudança. Quaisquer outros arquivos dentro de um aplicativo pode ser visto através da criação de uma função de manipulador usando ``` @ app_watch_handler``` decorador ````.

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

Dois exemplos de seu uso são relatados agora. Não se preocupe se você não undestand-los totalmente: o ponto-chave aqui é que mesmo código não-python pode ser recarregado automaticamente se explicita-lo com o ``` @ app_watch_handler``` decorador.

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 passado para `@ app_watch_handler` decorador deve ser relativo a um aplicativo. arquivos Python (ou seja, `*.py`) em uma lista passada para o decorador são ignorados uma vez que eles são vigiados por padrão. parâmetro da função de manipulador é uma lista de caminhos de arquivos que foram alterados. Todas as exceções dentro manipuladores são impressos em terminal.





---

## 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.

Ao processar qualquer solicitações HTTP existem algumas operações opcionais que pode desejar executar. Por exemplo parse o cookie para procurar informações sessão, confirmar uma transação de banco de dados, determinar o idioma preferido do cabeçalho HTTP e lookup internacionalização adequada, etc. Essas operações são opcionais. Algumas ações precisam deles e algumas ações não. Eles também podem depender um do outro. Por exemplo, se as sessões são armazenadas no banco de dados e nossa ação precisa dele, talvez seja necessário analisar o cookie de sessão a partir do cabeçalho, pegar uma conexão do pool de conexão do banco de dados, e - após a ação foi executada - salvar a sessão de volta no banco de dados se os dados foram alterados.

Fixtures PY4WEB fornecem um mecanismo para especificar o que uma ação necessidades para que py4web pode realizar as tarefas necessárias (e ignorar os não necessários) da maneira mais eficiente. Fixtures tornar o código eficiente e reduzir a necessidade de código clichê.

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 vem com alguns jogos pré-definidos para as ações que precisam sessões, conexões de banco de dados, internacionalização, autenticação e templates. A sua utilização será explicado neste capítulo. O desenvolvedor também é livre para adicionar Fixtures, por exemplo, para lidar com uma terceira língua template do partido ou a lógica sessão de terceiros.

### 6.1 Importante sobre Fixtures

Nos exemplos a seguir vamos explicar como aplicar fixtures individuais. Na prática fixtures podem ser aplicadas em grupos. Por exemplo:

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

Então você pode aplicar todo o ao mesmo tempo com:

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

## 6.2 Templates Py4web

PY4WEB, por padrão utiliza a linguagem de template yatl e fornece um dispositivo elétrico para ele.

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

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

Nota: Este exemplo assume que você criou o aplicativo da App andaimes, para que o index.html template já é criado para você.

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='[[ ]]'))
```

Observe que arquivos de template py4web são armazenados em cache na memória RAM. O objecto py4web cache é descrito mais tarde.

## 6.3 Sessões

O objeto da sessão é também um dispositivo elétrico. Aqui está um exemplo típico de uso para implementar um contador.

```
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
```

Observe que o objeto da sessão tem a mesma interface como um dicionário Python.

Por padrão, o objeto da sessão é armazenado em um cookie chamado, assinadas e criptografadas, utilizando o segredo fornecido. Se as mudanças secretas as sessões existentes são invalidados. Se o usuário muda de HTTP para HTTPS ou vice-versa, a sessão do usuário é invalidado. Sessão em cookies tem um pequeno limite de tamanho (4Kbytes depois de ser serializados e criptografados) para não colocar demasiada para eles.

Em py4web sessões são dicionários, mas eles são armazenados usando JSON (JWT especificamente),

portanto, você só deve armazenar objetos que são JSON serializado. Se o objeto não é JSON serializado, ele vai ser serializado usando o `` operador `__str__` e algumas informações podem ser perdidas.

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')
```

- Aqui `` algorithm`` é o algoritmo a ser usado para a assinatura de token JWT.
- `` Storage`` é um parâmetro que permite especificar um método alternativo de armazenamento de sessão (por exemplo, redis, ou base de dados).
- `` Same\_site`` é uma opção que impede CSRF ataques e é ativado por padrão. Você pode ler mais sobre ele *aqui* <<https://www.owasp.org/index.php/SameSite>> \_\_.

### 6.3.1 Sessão em memcache

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

Observe que um segredo não é necessária quando o armazenamento de cookies no memcache porque neste caso o cookie contém apenas o UUID da sessão.

### 6.3.2 Sessão em Redis

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

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.

### 6.3.3 Sessão no banco de dados

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

Um segredo não é necessária quando o armazenamento de cookies no banco de dados porque, neste caso, o cookie contém apenas o UUID da sessão.

Além disso, este é um caso em que o fixture uma (sessão) requer um outro suporte (db). Isso é feito automaticamente pelo py4web e os seguintes são equivalentes:

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

### 6.3.4 Sessão em qualquer lugar

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'))
```

Deixamos-lhe como um exercício para implementar validade, limitar o número de arquivos por pasta usando subpastas, e implementar o bloqueio de arquivos. No entanto, nós não recoment armazenar sessões no sistema de arquivos: é ineficiente e não escala bem.

## 6.4 Tradutor

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'))
```

A string ‘Olá mundo` será traduzido com base no arquivo de internacionalização em ‘traduções’ especificados pasta que melhor corresponde ao HTTP ``aceitar-language`` cabeçalho.

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 acção 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))

```

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"
 }
}

```

Ao visitar este site com o navegador preferência de idioma definida para Inglês e recarregar várias vezes você receberá as seguintes mensagens:

```

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"
 }
}

```

e definir preferência seu navegador para Italiano.

## 6.5 O fixture flash

É comum querer exibir “alertas” para os suers. Aqui nos referimos a eles como mensagens de flash. Há um pouco mais do que apenas exibir uma mensagem para o ponto de vista porque as mensagens flash podem ter estado que deve ser preservado após o redirecionamento. Também podem ser gerados lado servidor e do lado do cliente, pode haver apenas um no momento, eles podem ter um tipo, e eles devem ser

descartável.

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:

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

Ao definir o valor da mensagem do ajudante flash, uma variável flash é retornado pela ação e o gatilho os JS no template para injetar a mensagem na ``DIV # py4web-flash`` que você pode posicionar a sua conveniência. Também a classe opcional é aplicada ao HTML injetado.

Se uma página é redirecionada depois de um flash está definido, o flash é lembrado. Isto é conseguido por pedir o navegador para manter a mensagem temporariamente em um cookie de uma só vez. Depois de redirecionamento a mensagem é enviada de volta pelo navegador para o servidor e os conjuntos de servidor ele novamente automaticamente antes de retornar o conteúdo, a menos que seja substituído por um outro conjunto.

O cliente também pode definir / adicionar mensagens flash chamando:

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

defaults py4web para uma classe de alerta chamado ``default`` ea maioria dos quadros CSS definir classes para alertas chamado ``success``, ``error``, ``warning``, ``default``, e ``info``. No entanto, não há nada em py4web que hardcodes esses nomes. Você pode usar seus próprios nomes de classe.

## 6.6 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, por padrão, usa o PyDAL (Python abstração de dados Layer) que está documentado em um capítulo posterior. Aqui está um exemplo, lembre-se de criar o ``databases`` pasta em seu projeto caso ele não existe:

```
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"
```

Observe que as define fixação de banco de dados (cria / recria tabelas) automaticamente quando começa py4web (e cada vez que recarrega este app) e pega uma conexão do pool de conexão em cada solicitação HTTP. Além disso, cada chamada para o índice ``()`` ação é envolvido numa transacção e que compromete ``on\_success`` e reverte ``on\_error``.

## 6.7 Advertências sobre Fixures

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

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

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

Nota thas este código só será capaz de exibir um formulário, para processá-lo depois de apresentar, necessidades código adicional para ser adicionado, como veremos mais tarde. Este exemplo supõe que você criou o aplicativo da App andaimes, de modo que um generic.html já é criado para você.

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.

## 6.8 Fixtures personalizados

Um fixture é um objecto com a seguinte estrutura mínima:

```
from py4web import Fixture

class MyFixture(Fixture):
    def on_request(self): pass
```

```
def on_success(self): pass
def on_error(self): pass
def transform(self, data): return data
```

se uma ação usa este equipamento:

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

Em seguida, `on_request()` é garantida para ser chamado antes do índice `()` função é chamada. A `on_success()` é garantida para ser acionado se os índice `()` função retorna com sucesso ou aumentos `HTTP` ou executa um `redirect`. A `on_error()` é garantida para ser chamado quando o índice `()` função levanta qualquer outra exceção para além `HTTP`. A função `transform` é chamado para executar qualquer transformação desejada do valor devolvido pelo índice `()` função.

## 6.9 Auth e Auth.user

`Auth` e `auth.user` são ambos os jogos. Eles dependem de `session`. O papel do acesso é fornecer a ação com informações de autenticação. Ele é utilizado como segue:

```
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).

`Auth.enable()` registadoras múltiplas ações incluindo `{nomeaplic} / auth / login` e que exige a presença de a `auth.html` molde e a `auth` componente valor fornecido pela o aplicativo `_scaffold`.

O objeto `auth` é a fixure. Ele gerencia as informações do usuário. Ela expõe um único método:

```
auth.get_user()
```

que retorna um dicionário pitão contendo a informação do utilizador actualmente em sessão. Se o usuário não está conectado, ele retorna `None`. O código do exemplo redireciona para a `auth / login` A página se não houver usuário.

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')
```

Este fixture redireciona automaticamente para a página login ``auth /`` se o usuário não está conectado. Depende de ``auth``, que depende de ``db`` e ``session``.

A ``fixação Auth`` é baseado plug-in e suporta vários métodos de plug-in. Eles incluem oauth2 (Google, Facebook, Twitter), PAM, LDAP, e SMAL2.

Aqui está um exemplo de como usar o plugin Google oauth2:

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

A ``client\_id`` e ``client\_secret`` são fornecidos pelo Google. O URL de retorno é a opção padrão para py4web e deve ser na lista de autorizações com o Google. Todos os ``plugins Auth`` são objetos. Diferentes plugins são configurados de diferentes maneiras, mas eles são registrados usando ``auth.register\_plugin(...)``. Os exemplos são fornecidos no ``\_scaffold / common.py``.

## 6.10 Caching e Memoize

py4web fornece um cache no objeto ram que implementa a Última Recently Used (LRU) Algoritmo. Ele pode ser usado para armazenar em cache qualquer função através de um decorador:

```
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())
```

Ele irá armazenar em cache (memoize) o valor de retorno da função ``hello``, como função da entrada ``name``, durante até 60 segundos. Ele irá armazenar em cache de 1000 valores mais usados recentemente. Os dados são sempre armazenados na memória RAM.

O objeto de cache não é um dispositivo elétrico e não deve e não pode ser registrado usando o ``@action.uses`` objeto, mas nós mencioná-lo aqui, porque alguns dos dispositivos elétricos de usar esse objeto internamente. Por exemplo, arquivos de template são armazenadas em cache na memória RAM para evitar acessar o sistema de arquivo de cada vez que um template precisa ser renderizado.

## 6.11 Decoradores de conveniência

A ``aplicação \_scaffold``, no ``define common.py`` dois decoradores especiais conveniennce:

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

e

```
`@Authenticated def index (): retorno Dict ()`
```

Aplicam-se todos os decoradores abaixo, utilizar um template com o mesmo nome que a função (.html), e também registrar-se uma via com o nome de acção seguido o número de argumentos da acção separadas por uma barra (/).

@unauthenticated não requer que o usuário seja identificado. @authenticated necessário que o usuário estar logado.

Se pode ser combinado com (e preceder) outro `` @ action.uses (...) `` , mas eles não devem ser combinados com `` @Action (...) `` , porque eles executar essa função automaticamente.

---

## A camada de abstracção de banco de dados (DAL)

---

### 7.1 Dependências

py4web vem com uma camada de abstracção de banco de dados (DAL), uma API que mapeia Python objetos no banco de dados objetos, como consultas, tabelas e registros. A DAL gera dinamicamente o SQL em tempo real usando o dialeto especificado para o back-end de banco de dados, de modo que você não precisa de código SQL gravação ou aprender diferentes dialetos SQL (SQL termo é utilizado genericamente), ea aplicação irá ser portátil entre diferentes tipos de bancos de dados. Uma lista parcial de bases de dados suportadas é mostrado na tabela abaixo. Por favor, verifique no site py4web e lista de endereços para adaptadores mais recentes. Google NoSQL é tratado como um case particular.

A seção Gotchas no final deste capítulo tem mais algumas informações sobre bancos de dados específicos.

A distribuição binária Windows funciona fora da caixa com SQLite, MSSQL, PostgreSQL e MySQL. A distribuição binária Mac funciona fora da caixa com SQLite. Para usar qualquer outro banco de dados back-end, executado a partir da distribuição fonte e instalar o driver apropriado para o back-end necessário.

Uma vez que o driver apropriado está instalado, inicie py4web a partir da fonte, e vai encontrar o driver. Aqui está uma lista de drivers py4web pode usar:

banco de dados	motoristas (fonte)
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

``Sqlite3``, ``pymysql``, e ``navio imaplib`` com py4web. Apoio do MongoDB é experimental. A opção IMAP permite usar DAL para acesso IMAP.

## 7.2 A DAL: Um rápido passeio

define py4web as seguintes classes que compõem o DAL:

O **DAL** objeto representa uma conexão de banco de dados. Por exemplo:

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

**Tabela** representa uma tabela de banco de dados. Você faz Tabela não diretamente instanciar; em vez disso, ``DAL.define\_table`` instancia-lo.

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

Os métodos mais importantes de uma tabela são:

``Insert``, ``truncate``, ``drop``, e ``import\_from\_csv\_file``.

**Campo** representa um campo de banco de dados. Ele pode ser instanciado e passado como um argumento para ``DAL.define\_table``.

**Linhas DAL** é o objeto retornado por um banco de dados selecionado. Ele pode ser pensado como uma lista de ``linhas row``:

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

**Row** contém valores de campo.

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

**consulta** é um objeto que representa um SQL cláusula “where”:

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

**Set** é um objeto que representa um conjunto de registros. Seus métodos mais importantes são ``count``, ``SELECT``, ``update``, e ``DELETE``. Por exemplo:

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

**Expressão** é algo como um ``orderby`` ou ``expressão groupby``. A classe campo é derivado da expressão. Aqui está um exemplo.

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

## 7.3 Usando o DAL “stand-alone”

A DAL pode ser usado num ambiente não-py4web através

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

## 7.4 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
```

A seqüência de conexão é chamado de ``\_uri`` porque é uma instância de um 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.

### 7.4.1 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)
```

## 7.4.2 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', pool_size=0)
```

``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"):

*** 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``
Oráculo	`` Oracle: // username / password @ test``
*** DB2	`` Db2: // username: password @ test``
*** Ingres	`` Ingres: // username: password @ localhost / test``
*** Sybase	`` Sybase: // username: password @ localhost / test``
*** Informix	`` Informix: // username: password @ test``
*** Teradata	`` Teradata: // DSN = dsn; UID = utilizador; PWD = pass; DATABASE = test``
*** CUBRID	`` CUBRID: // username: password @ localhost / test``
*** SAPDB	`` Sapdb: // username: password @ localhost / test``
*** IMAP	`` Imap: // utilizador: senha @ servidor: port``
*** MongoDB	`` Mongod: // username: password @ localhost / test``
*** Google / SQL **	`` Google: sql: // project: instance / database``
*** Google / NoSQL **	`` Google: datastore``
*** Google / NoSQL / NDB **	`` Google: armazenamento de dados + ndb``

Observe que em SQLite banco de dados consiste em um único arquivo. Se ele não existir, ele será criado. Este arquivo está bloqueado cada vez que é acessado. No caso do MySQL, PostgreSQL, MSSQL, Firebird, Oracle, DB2, Ingres e Informix o "teste" de banco de dados deve ser criado py4web fora. Uma vez que a conexão é estabelecida, py4web irá criar, alterar e eliminar tabelas de forma adequada.

? Na sequência de conexão MySQL, o `` set\_encoding = utf8mb4`` no final define a codificação para UTF-8 e evita um `` inválido sequência de caracteres utf8: `` erro em caracteres Unicode que consistem em quatro bytes, como por padrão , MySQL só pode lidar com caracteres Unicode que consistem de uma de três bytes.

No caso Google / NoSQL os `` + voltas opção ndb`` no NDB. NDB usa um buffer Memcache para ler dados que são acessados frequentemente. Isto é completamente automático e feito no nível de armazenamento de dados, não no nível py4web.

Também é possível definir a sequência de conexão para `` None``. Neste caso DAL não vai conectar a qualquer banco de dados back-end, mas a API ainda pode ser acessado para testar.

Algumas vezes pode ser necessário para gerar SQL como se você tivesse uma conexão, mas sem realmente conectar ao banco de dados. Isso pode ser feito com

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

Neste case, você será capaz de chamar ``\_select``, ``\_insert``, ``\_update``, e ``\_delete`` para gerar SQL, mas não chamar ``SELECT``, ``insert``, ``update``, e ``DELETE``. Na maioria dos cases, você pode usar ``do\_connect = False`` mesmo sem ter os drivers de banco de dados necessárias.

Observe que, por padrão py4web usou 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.

### 7.4.3 O pool de conexões

Um argumento comum do construtor DAL é a ``pool\_size``; o padrão é zero.

Como é bastante lento para estabelecer uma nova conexão com o banco para cada solicitação, implementos py4web um mecanismo para o pool de conexão. Uma vez que uma conexão é estabelecida e a página foi servido ea transação concluída, a conexão não é fechada, mas vai para uma piscina. Quando o próximo pedido http chega, py4web tenta reciclar uma conexão do pool e uso que para a nova transação. Se não há conexões disponíveis na piscina, uma nova conexão é estabelecida.

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.

### 7.4.4 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 case 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.

### 7.4.5 Tabelas preguiçosos

ajuste ``lazy\_tables = True`` fornece um grande aumento de desempenho. Veja abaixo: *tabelas preguiçosos*  
<#lazy\_tables> \_\_

### 7.4.6 Aplicativos de modelo-less

Usando o diretório model de py4web para seus modelos de aplicação é muito conveniente e produtiva. Com mesas preguiçosos e modelos condicionais, o desempenho é normalmente aceitável, mesmo para grandes aplicações. Muitos desenvolvedores experientes usar isso em ambientes de produção.

No entanto, é possível definir tabelas DAL sob demanda dentro de funções de controlador ou módulos. Isso pode fazer sentido quando o número ou a complexidade das definições de tabela sobrecarrega o uso de tabelas preguiçosos e modelos condicionais.

Isto é referido como “modelo-less” desenvolvimento pela comunidade py4web. Isso significa menos uso da execução automática de arquivos Python no diretório model. Isso não implica o abandono do conceito de modelos, visualizações e controladores.

A auto-execução de código Python do PY4WEB dentro do diretório model faz isso para você:

- modelos são executados automaticamente cada vez que um pedido é processado
- modelos acessar escopo global do py4web.

Modelos também para fazer sessões de shell interativo úteis quando py4web é iniciado com a opção -M comando.

Além disso, lembre manutenção: outros desenvolvedores py4web esperar encontrar definições de modelo no diretório de modelo.

Para usar o “-modelo menos” abordagem, você assumir a responsabilidade de fazer essas duas tarefas domésticas. Você chama as definições de tabela quando você precisar deles, e proporcionar o acesso necessário passado como parâmetro.

Por exemplo, uma aplicação típica modelo de menos pode deixar as definições dos objetos de conexão de banco de dados no arquivo de modelo, mas definir as tabelas na demanda por função de controlador.

O case típico é mover as definições de tabela para um arquivo de módulo (um arquivo Python salvos no diretório de módulos).

Se a função de definir um conjunto de tabelas é chamado `define_employee_tables()` em um módulo chamado `table_setup.py`, seu controlador que quer consulte as tabelas relacionadas a registros de funcionários, a fim de fazer uma SQLFORM precisa chamar os `define_employee_tables()` função antes de acessar qualquer tabela. Os `define_employee_tables()` necessidades de função para aceder ao objecto de ligação da banco de dados, a fim de definir tabelas. É necessário passar o objecto db para os `define_employee_tables()` (como mencionado acima).

## 7.4.7 Bancos de dados replicados

O primeiro argumento de `DAL(...)` pode ser uma lista de URIs. Neste case py4web tenta se conectar a cada um deles. O objetivo principal para isso é que lidar com vários servidores de banco de dados e distribuir a carga de trabalho entre eles). Aqui está um case de uso típico:

```
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.

## 7.4.8 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.

O nome do adaptador é o mesmo que o utilizado na strings de ligação DAL. Então, se você quiser verificar contra PostgreSQL e MSSQL, em seguida, sua sequência de conexão ficaria da seguinte forma:



```
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.

Para back-ends suportados você também pode especificar se você gostaria de verificar contra as palavras-chave SQL não reservados bem. Neste case, você poderia acrescentar ``\_nonreserved`` ao nome. Por exemplo:

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

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

<b>** ** PostgreSQL</b>	<b>`` Postgres (_nonreserved) ``</b>
<b>** ** MySQL</b>	<b>`` Mysql``</b>
<b>** ** FireBird</b>	<b>`` Firebird (_nonreserved) ``</b>
<b>** ** MSSQL</b>	<b>`` mssql``</b>
<b>Oráculo</b>	<b>`` oracle``</b>

## 7.4.9 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
```

## 7.4.10 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.

### 7.4.11 Outros parâmetros do construtor DAL

#### Local de pasta do banco de dados

``Folder`` define o lugar onde migração arquivos serão criados (ver *Migrações* <#table\_migrations> \_\_ seção neste capítulo para mais detalhes). Ele também é usado para bancos de dados SQLite. Definida automaticamente entre py4web. Definir um caminho ao utilizar DAL fora py4web.

#### Configurações padrão de migração

As configurações de migração construtor DAL são booleans que afetam padrões e comportamento global.

``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

### 7.4.12 Experimentar com o shell py4web

Você pode experimentar com a API DAL usando o shell py4web, que está disponível usando o comando ``shell`` (leia mais no *Capítulo 1* <# capítulo 01 # command\_line\_options> \_\_).

Você precisa escolher uma aplicação para executar o shell on, mente que as mudanças de banco de dados pode ser persistente. Portanto, tenha cuidado e NÃO exite para criar uma nova aplicação para fazer testes em vez de adulteração de um já existente.

Comece por criar uma conexão. Por uma questão de exemplo, você pode usar SQLite. Nada nesta discussão muda quando você alterar o motor de back-end.

Note-se que a maior parte dos trechos de código que contêm o python prompt de ``>>>`` estão diretamente executável através de um shell simples, que você pode obter usando ``opções de linha de comando -PS``.

## 7.5 Construtor Table

As tabelas são definidos na DAL via ``define\_table``.

### 7.5.1 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: `` rname``, `` redefinir``, `` 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.

## 7.5.2 ``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``.

## 7.5.3 ``Plural`` e singular``

Como pydal é uma DAL geral, ele inclui atributos singular e plural para se referir aos nomes de tabela para que os elementos externos podem usar o nome apropriado para uma mesa. Um case de uso está em web2py com objetos SmartGrid com referências a tabelas externas.

## 7.5.4 ``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.

## 7.5.5 ``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')
```

O atributo formato será usado para duas finalidades: - Para representar registros referenciados em select / opção menus suspensos. - Para definir o atributo ``db.othertable.otherfield.represent`` para todos os

campos que referenciam esta tabela. Isto significa que o ``construtor Form`` não vai mostrar referências de ID mas vai usar a representação formato preferido vez.

### 7.5.6 ``Rname``: nome real

``Rname`` define um nome de back-end de banco de dados para a tabela. Isso faz com que o nome da tabela py4web um alias, e ``rname`` é o verdadeiro nome usado ao construir a consulta para o backend. Para ilustrar apenas um uso, ``rname`` pode ser usado para fornecer MSSQL nomes de tabela totalmente qualificado acessar as tabelas pertencentes a outras bases de dados no servidor: ``rname = "db1.dbo.table1"``: python

### 7.5.7 ``Primarykey``: Suporte para tabelas legadas

``Primarykey`` ajuda tabelas legadas de apoio com chaves primárias existentes, até mesmo multi-parte. Veja *bancos de dados legados e tabelas com chave* <#LegacyDatabases> \_\_ neste capítulo.

### 7.5.8 ``Migrate``, ``fake\_migrate``

``Migrate`` define migração opções para a mesa. Consulte *Migrações* <#table\_migrations> \_\_ seção neste capítulo para obter detalhes.

### 7.5.9 ``Table\_class``

Se você definir sua própria classe Table como uma sub-classe de pydal.objects.Table, você pode fornecê-la aqui; isso permite-lhe ampliar e métodos de substituição. Exemplo:

```
from pydal.objects import Table

class MyTable(Table):
    ...

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

### 7.5.10 ``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 seqüências personalizadas.

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

### 7.5.11 ``Trigger\_name``

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

### 7.5.12 ``polymodel``

Para o Google App Engine

### 7.5.13 ``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 ``requires`` poderiam ser adicionados às definições de campo e a 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``.

### 7.5.14 Tabelas de descanso, um grande aumento de desempenho

modelos py4web são executados antes de controladores, por isso, todas as tabelas são definidas em cada solicitação. Nem todas as tabelas são necessários para lidar com cada pedido, por isso é possível que algum do tempo gasto definindo tabelas é desperdiçado. modelos condicionais (veja *Model-menos aplicações* <#model\_less\_applications> \_\_) pode ajudar, mas ofertas py4web um grande aumento de desempenho através lazy\_tables. Este meio de recursos que a criação da tabela é adiada até que a tabela é realmente referenciado. Permitindo tabelas preguiçosos é feita quando inicializar um banco de dados via o construtor DAL. Ele requer definir os lazy\_tables parâmetro: ``DAL(..., lazy\_tables = True)``: python Este é um dos impulsos mais significativos de desempenho em tempo de resposta em py4web.

### 7.5.15 Adicionando atributos para campos e tabelas

Se você precisa adicionar atributos personalizados aos campos, você pode simplesmente fazer isso: ``db.table.field.extra = {}``

“Extra” não é uma palavra-chave; é um atributos personalizados agora ligados ao objeto de campo. Você pode fazê-lo com mesas também, mas eles devem ser precedidos por um sublinhado para evitar conflitos de nomes com campos:

``Db.table.\_extra = {}``: pitão

## 7.6 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, uploaddfs=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 especifique 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 caso, 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` é um validador ou uma lista de validadores. Isto não é usado pelo DAL, mas ele é usado por `Form`. Os validadores predefinidos para os tipos de dados são mostrados na próxima seção.

Note-se que enquanto `requer = ...` é aplicada ao nível de formas, `required = True` é aplicada ao nível da DAL (inserção). Além disso, `notnull`, `unique` e `ondelete` são aplicadas ao nível da banco de dados. Enquanto eles, por vezes, pode parecer redundante, é importante manter a distinção ao programar com o 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` só se aplica aos campos do tipo “Upload”. Um campo de lojas do tipo “Carregar” o nome de um arquivo salvo em outro lugar, por padrão no sistema de arquivos na pasta “uploads /” aplicação. Se `uploadfield` é definida como `True`, em seguida, o arquivo é armazenado em um campo blob dentro da mesma tabela eo valor de `uploadfield` é o nome do campo blob. Isso será discutido com mais detalhes posteriormente no \* Mais informações sobre os envios \* neste capítulo.
- `Uploadfolder` define a pasta para arquivos enviados. Por padrão, um arquivo carregado vai para a pasta da aplicação “uploads /”, que é em `os.path.join(request.folder, “uploads”)` (isto não parece ser o caso para `MongoAdapter` no presente). Por exemplo: `Field(..., uploadfolder = os.path.join(request.folder, “static / temp”))`: python vai fazer upload de arquivos para a “py4web / aplicativos / myapp / static / temp” pasta.
- `Uploadseparate` Se definido como verdadeiro vai fazer upload de arquivos em diferentes subpastas da \* `uploadfolder` pasta \*. Esta é otimizado para evitar muitos arquivos na mesma pasta / subpasta. ATENÇÃO: Não é possível alterar o valor de `uploadseparate` de `True` para `False` sem quebrar links para uploads existentes. `py4web` ou usa as subpastas separadas ou não. Alterar o comportamento após arquivos foram enviados impedirá `py4web` de ser capaz de recuperar esses arquivos. Se isso acontece, é possível mover arquivos e corrigir o problema, mas isso não é descrito aqui.

- ``Uploadfs`` permite que você especifique 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`` determina se o arquivo enviado correspondente deve ser suprimido quando o registro referenciando o arquivo é excluído. Por apenas os campos "Upload". No entanto, os registros excluídos pelo próprio banco de dados devido a uma operação em cascata não vai autodelete gatilho de py4web. O grupo Google py4web tem solução discussões.
- ``Widget`` deve ser um dos objetos de widgets disponíveis, incluindo widgets personalizados, por exemplo: ``SQLFORM.widgets.string.widget``. Uma lista de widgets disponíveis será discutido mais tarde. Cada tipo de campo tem um widget padrão.
- ``Label`` é uma string (ou um ajudante ou algo que pode ser serializado para uma string) que contém o rótulo a ser usado para este campo em formas geradas automaticamente.
- ``Comment`` é uma string (ou um ajudante ou algo que pode ser serializado para uma string) que contém um comentário associado a este campo, e será exibido à direita do campo de entrada nos formulários gerados automaticamente.
- ``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.
- ``Searchable`` declara se um campo é pesquisável em grades (``SQLFORM.grid`` e ``SQLFORM.smartgrid`` são descritos em \* Capítulo 7 ../07\*). Observe que um campo também deve ser legível a ser pesquisado.
- ``Listable`` declara se um campo é visível em grades (quando listando vários registros)
- ``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.
- ``Represent`` pode ser Nenhum ou pode apontar para uma função que recebe um valor de campo e retorna uma representação alternativa para o valor do campo. Exemplos:

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

- ``Filter\_in`` e ``filter\_out`` pode ser configurado para chamáveis para processamento adicional do valor de campo. ``Filter\_in`` é passado o valor do campo a serem gravados para o banco de dados antes de uma inserção ou atualização enquanto ``filter\_out`` é passado o valor recuperado do banco de dados antes da atribuição de campo. O valor retornado pelo que pode ser chamado é então utilizado. Veja *filter\_in* e *filter\_out* <#filter\_in\_filter\_out> \_\_ neste capítulo.
- ``Custom\_qualifier`` é um qualificador SQL personalizado para o campo para ser usado na criação da tabela (não pode usar para campo do tipo "id", "referência" ou "big-referência").



## 7.6.1 Tipos de campo

tipo de campo	** validadores de campo padrão **
<code>String</code>	<code>IS_LENGTH</code> (comprimento) `` comprimento padrão é 512
<code>text</code>	<code>IS_LENGTH</code> (comprimento) `` comprimento padrão é 32.768
<code>blob</code>	Comprimento padrão <code>None</code> é <code>2 ** 31</code> (2 GIB)
<code>boolean</code>	<code>None</code>
<code>integer</code>	<code>IS_INT_IN_RANGE</code> ( <code>-2 ** 31, 2 ** 31</code> ) ``
<code>Double</code>	<code>IS_FLOAT_IN_RANGE</code> ( <code>-1e100, 1e100</code> ) ``
<code>Decimal (n, m)</code> ``	<code>IS_DECIMAL_IN_RANGE</code> ( <code>-10 ** 10, 10 ** 10</code> ) ``
<code>date</code>	<code>IS_DATE</code> () ``
<code>Time</code>	<code>IS_TIME</code> () ``
<code>datetime</code>	<code>IS_DATETIME</code> () ``
<code>password</code>	<code>IS_LENGTH</code> (comprimento) `` comprimento padrão é 512
<code>Upload</code>	Comprimento padrão é 512 <code>None</code>
<code>Referência &lt;table&gt;</code> ``	<code>IS_IN_DB</code> (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</code> (d b, table._id, formato, múltiplos = <code>TRUE</code> ) ``
<code>json</code>	<code>IS_EMPTY_OR</code> ( <code>IS_JSON</code> ()) `` comprimento padrão é 512
<code>bigint</code>	<code>IS_INT_IN_RANGE</code> ( <code>-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.

O `` lista: <type>`` campos são especiais porque eles são projetados para tirar proveito de certas características Desnormalização on NoSQL (no case do Google App Engine NoSQL, os tipos de campo `` `ListProperty``` e `` `StringListProperty``` ) e volta-porta-lhes todas as outras bases de dados relacionais suportados. Em bancos de dados relacionais listas são armazenados como um `` `text``` campo. Os itens são separados por um `` `|``` e cada `` `|``` no item corda escapou, como um `` `||```. Eles são discutidos na lista \*: e contém seção \* neste capítulo.

O `` tipo de campo `json``` é bastante explicativo. Ele pode armazenar qualquer objeto JSON serializado. Ele é projetado para trabalhar especificamente para MongoDB e portadas para os outros adaptadores de banco de dados para portabilidade.

`` Campos `blob``` também são especiais. Por padrão, os dados binários são codificados em Base64 antes de ser armazenado no campo banco de dados real, e é decodificado quando extraído. Isto tem o efeito negativo de utilizar 33% mais espaço de armazenagem do que o necessário em campos `blob`, mas tem o advantageof fazendo a comunicação independente de convenções específico de back-end-escapam.



## 7.6.2 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.

## 7.7 Migrações

``Controlos define\_table`` se existe ou não a tabela correspondente. Se isso não acontecer, ele gera o SQL para criá-lo e executa o SQL. Se a tabela existe, mas difere do que está sendo definido, gera o SQL para alterar a tabela e executa-lo. Se um campo mudou tipo, mas não o nome, ele vai tentar converter os dados (se você não quer isso, você precisa redefinir a tabela duas vezes, a primeira vez, deixando py4web largar o campo removendo-o, e pela segunda vez adicionar o campo recém-definido para que py4web pode criá-lo.). Se a tabela existe e coincide com a definição atual, ele vai deixá-lo sozinho. Em todos os casos ele irá criar o ``db.person`` objeto que representa a mesa.

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

Observe que por py4web padrão usa a pasta “app / bancos de dados” para o arquivo de log e todos os outros arquivos de migração de que necessita. Você pode alterar essa configuração o argumento ``folder`` para DAL. Para definir um nome de arquivo de log diferente, por exemplo “migrate.log” você pode fazer ``db = DAL (... , adapter\_args = dict (logfile = “migrate.log”))``: python

O primeiro argumento de ``define\_table`` é sempre o nome da tabela. Os outros argumentos sem nome são os campos (campo). A função também tem um argumento palavra-chave opcional chamado “migrar”:

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

O valor de migrar é o nome do arquivo onde as informações lojas py4web 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 descartado eo arquivo correspondente ainda existem, ele pode ser removido manualmente. Por padrão, migre é definida como True. Este causas py4web para gerar o nome do arquivo a partir de um hash da string de conexão. Se migre 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-se que só migra py4web novas colunas, as colunas removido, e mudanças no tipo de coluna (exceto no SQLite). py4web não altere de migram em atributos tais como alterações nos valores de default: ``unique``, ``notnull``, e ``ondelete``.

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

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

Este é o comportamento recomendado quando dois aplicativos compartilham o mesmo banco de dados. Apenas um dos dois aplicativos devem executar migrações, o outro deve desativá-las.

### 7.7.1 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.

Antes de tentar correção de migração problemas é prudente fazer uma cópia de <quotechar> aplicações / yourapp / databases / \*. Mesa <quotechar> arquivos.

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.

### 7.7.2 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
```

## 7.8 ``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.

## 7.9 `` `` commit`` e rollback``

Inserir, truncado, excluir e operações de atualização não estão realmente comprometidos até que as questões py4web o comando cometer. A criar e operações de gota pode ser executado imediatamente, dependendo do motor de banco de dados. Chamadas para ações py4web são automaticamente envolvido em transações. Se você executou os comandos da shell, você é obrigado a cometer manualmente:

```
>>> 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
```

Código em modelos, visualizações e controladores está incluído no código py4web parecida com esta (pseudo-código):

```
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
```

Assim, em modelos, visualizações e controladores não há necessidade de chamar cada vez ``commit`` ou ``rollback`` explicitamente em py4web a menos que você precisa um controle mais granular. No entanto, em módulos que você vai precisar usar ``commit``.

## 7.10 Raw SQL

### 7.10.1 Temporização de consultas

Todas as consultas são automaticamente cronometrado por py4web. A variável ``db.\_timings`` é uma lista de tuplos. Cada tupla contém a consulta SQL crus como passados para o driver de banco de dados e o tempo que levou para executar em segundos. Esta variável pode ser exibida em vistas usando a barra de ferramentas:

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

### 7.10.2 ``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.

Se ``as\_dict`` é definida como True, os resultados do cursor retornado pelo driver DB será convertido em uma sequência de dicionários introduzidos com os nomes dos campos db. Os resultados retornados com

`as_dict = True` são os mesmos que os devolvido ao aplicar `as_list()` para um select normal.:

```
[{'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.

### 7.10.3 `_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 não gera consultas usando o `*` operador. py4web é sempre explícita ao selecionar campos.

## 7.11 `Drop`

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

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

## 7.12 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.

## 7.13 Bancos de dados legados e tabelas com chave

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

A maneira mais fácil é quando essas condições forem atendidas: - Cada tabela deve ter um campo inteiro de auto-incremento exclusivo chamado “id” - Os registros devem ser referenciado exclusivamente usando o campo “id”.

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

Se a tabela de legado tem um campo inteiro de auto-incremento, mas não é chamado de “id”, py4web ainda pode acessá-lo, mas a definição da tabela deve declarar o campo de incremento automático com o tipo de “id” (que está usando campo `` (“ ... “, “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.

## 7.14 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 exem-

plo:

```
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-los separadamente. Isto significa que há uma possibilidade de que um dos commits sucede e uma falha. A transação distribuída impede que isso aconteça.

## 7.15 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)
```

## 7.16 ``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.

## 7.17 `` Comando SELECT``

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

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

Ele retorna um objecto de classe iteráveis ``pydal.objects.Rows`` cujos elementos são objectos da fileira. ``Pydal.objects.Row`` objetos agir como dicionários, mas os seus elementos também pode ser acedida como atributos, como ``gluon.storage.Storage``. The ex diferente do último porque os seus valores são apenas para leitura.

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 mesa 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]
```

você pode extrair seus valores usando várias expressões equivalentes:

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

O último sintaxe é particularmente útil quando se selecciona en expressão em vez de uma coluna. Vamos mostrar isso mais tarde.

Você também pode fazer

```
rows.compact = False
```

desativar a notação

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

e permitir que, em vez disso, a notação menos compacto:

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

Sim isso é incomum e raramente necessário.

Row objetos também tem dois métodos importantes:

```
row.delete_record()
```

e

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

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

Python “iterators” são um tipo de “preguiçoso-avaliação”. Eles dados ‘alimentar’ um passo de tempo; laços tradicionais Python criar todo o conjunto de dados na memória antes de looping.

O uso tradicional de selecionar é:

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

mas para um grande número de linhas, usando uma alternativa à base de iterador tem uso de memória dramaticamente inferior:

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

Testes mostram que isto é de cerca de 10% mais rápido, mesmo em máquinas com muita RAM.

### 7.17.2 Renderizando Rows com represent

Você pode querer reescrever linhas retornadas por seleção para tirar proveito de informações de formatação contida na representa a criação dos campos.

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

Se você não especificar um índice, você tem um gerador para iterar sobre todas as linhas:

```
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, ele retorna uma cópia transformada da linha original, então não há nenhuma update\_record (que você não iria querer de qualquer maneira) ou delete\_record.

### 7.17.3 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.

Nota: atalho inserção anteriormente era ``db.table[0] = ...``. Ele mudou em PyDAL 19,02 para permitir o uso normal de identificação 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.

### 7.17.4 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``.

### 7.17.5 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.

### 7.17.6 ``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).

```
>>> print 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

Estes estão envolvidos na gestão \* um para muitas relações \*. Eles são descritos em \* INNER JOIN \* e exterior \* LEFT JOIN \* seções \* respectivamente.

### cache, em cache

Um uso exemplo que dá muito mais rápido selecciona é: ``linhas = db(query).Select(cache = (cache.ram, 3600), cacheable = True)``: Olhar python na seção \* \* Caching selecciona neste capítulo, para entender o que os trade-offs são.

## 7.17.7 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'
```

### 7.17.8 “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.

### 7.17.9 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()
```

### 7.17.10 ``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
```

### 7.17.11 ``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': 2L, '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': 3L, 'name': 'Philip'}>
```

Note, você deve evitar o uso de ``row.update\_record()`` sem argumentos quando o objeto ``row``

contém campos que têm um atributo ``update`` (por exemplo, ``Field ("modified\_on", update = request.now)``). Chamando ``row.update\_record()`` irá reter \* todos \* os valores existentes no objeto ``row``, portanto, quaisquer campos com ``atributos update`` não terá nenhum efeito neste caso. Seja particularmente atento a isso com mesas que incluem ``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`.

### 7.17.12 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".

### 7.17.13 `` `` 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.

### 7.17.14 `` `` 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()
```

Estes métodos são convenientes para a passagem de linhas de views genéricas e ou armazena registros em sessões (uma vez que linhas objetos em si não pode ser serializado desde incluir uma referência a uma

conexão DB aberto):

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

### 7.17.15 Combinando Rows

Fileiras objectos 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
```

### 7.17.16 ``Find``, ``exclude``, ``sort``

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

## 7.18 Outros métodos

### 7.18.1 ``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')
```

### 7.18.2 ``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')
```

exceto que ele chama os validadores para os campos antes de realizar a inserção e fianças se a validação não passa. Se a validação não passa os erros podem ser encontradas em ``ret.errors``. ``Ret.errors`` mantém um mapeamento de valores-chave, onde cada chave é o nome do campo cuja validação falhou, e o valor da chave é o resultado do erro de validação (muito parecido com ``form.errors``). Se passar, o ID do novo registro é em ``ret.id``. Mente que normalmente validação é feita pela lógica de processamento de formulário para essa função é raramente necessária.

Similarmente

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

funciona muito da mesma forma como

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

exceto que ele chama os validadores para os campos antes de realizar a atualização. Note que ele só funciona se consulta envolve uma única tabela. O número de registros atualizados podem ser encontrados em ``ret.updated`` e erros será no ``ret.errors``.

## 7.19 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': 5L}>
```

Observe que o valor calculado é armazenado no banco de dados e não é computado na recuperação, como no case de campos virtuais, descrito a seguir. Duas aplicações típicas de campos computados são: - em aplicações wiki, para armazenar o texto de entrada wiki processado como HTML, para evitar re-processamento em cada solicitação - para a pesquisa, para calcular os valores normalizados para um campo, a ser usado para pesquisa.

Campos computados são avaliadas na ordem em que eles são definidos na definição da tabela. Um campo computado pode referir-se aos campos computados previamente definidos (novo após v 2.5.1)

## 7.20 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.

### 7.20.1 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: ...))
```

Mente que campos virtuais não têm os mesmos atributos como campos regulares (comprimento, padrão, exigida, etc). Eles não aparecem na lista de ``db.table.fields`` e em versões mais antigas do py4web eles exigem uma abordagem especial para exibição no SQLFORM.grid e SQLFORM.smart-grid. Veja a discussão sobre grades e campos virtuais em \*Capítulo 12\*.

## 7.20.2 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(object):
    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(object):
    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(object):
    def total_price(self):
        return self.item.unit_price * self.order_item.quantity

rows.setvirtualfields(order_item=MyVirtualFields())

for row in rows:
    print row.order_item.total_price

```

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

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

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

for row in rows:
    print row.order_item.discounted_total_price

```

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

```

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

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

ou mais curto utilizando uma função lambda:

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

## 7.21 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)>
```

Tabela “coisa” tem dois campos, o nome da coisa eo proprietário da coisa. O campo “owner\_id” é um campo de referência, pretende-se que a referência de campo outra tabela pelo seu id. Um tipo de referência pode ser especificada em duas formas equivalentes, quer: `` campo ( “OWNER\_ID”, “pessoa de referência”) ``: pitão ou: `` campo ( “OWNER\_ID”, db.person) ``: pitão

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
```

### 7.21.1 Inner joins

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.

### 7.21.2 Left outer join

Observe que Carl não aparecer 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``.

### 7.21.3 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.

## 7.22 Muitos para muitos

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
```

Uma alternativa mais leve para muitos-para-muitos relações é a marcação, encontram-se um exemplo disso na próxima seção. Marcação de obras, mesmo em backends de banco de dados que não suportam JOINS como o Google App Engine NoSQL.

## 7.23 Marcação de registros

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

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

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

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

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

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

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

Ele é implementado internamente como uma tabela com o nome: tags \*\*, que neste exemplo seria

db.thing\_tags\_default, porque nenhum caminho foi especificado nas etiquetas (tabela, path = "default") construtor

O método `find` está fazendo uma busca por `startswith` do caminho passado como parâmetro. Em seguida, encontrar ([ "cor"]) deve retornar id1 e ID2 porque ambos os registros têm etiquetas que começam com "cor". py4web utiliza tags como um mecanismo flexível para gerenciar permissões.

## 7.24 ``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 escapados 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.

## 7.25 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
```

### 7.25.1 `` Like``, `` ilike``, `` regexp``, `` startswith``, `` endswith``, `` contains``, `` up-



## per``, ``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%')
```

Lembre-se que ``contains`` tem um significado especial para ``lista`<type>`` campos, como discutido na lista anterior \*: e contém seção \*.

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 regexp`` 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`` métodos 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
```

### 7.25.2 ``Year``, ``month``, ``day``, ``hour``, ``minutes``, ``seconds``

A data ea data e hora campos têm ``day``, ``month`` e ``métodos 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
```

### 7.25.3 ``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 seccione 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 seleteo aninhada 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 aninhada 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.

### 7.25.4 ``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]
43
```

### 7.25.5 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])
```

### 7.25.6 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'))
```

## 7.26 Gerando SQL puro

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.

## 7.27 Exportar e importar dados

### 7.27.1 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()))
```

Note-se que a conversão de um objeto ``Rows`` em uma strings utilizando o Python 2 produz um utf8 codificado strings binária. Para obter uma codificação diferente você tem que perguntar para ele explicitamente, por exemplo, com:

``Unicode(str(db(db.person).Select()), “UTF-8”) codificar (...)``. Pythonn

Ou em Python 3:

```
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)
```

Mais uma vez, ao usar Python 3, você pode ser explicit sobre a codificação para o arquivo de exportação:

```
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)
```

Ao importar, py4web olha para os nomes de campo no cabeçalho CSV. Neste exemplo, ele encontra duas colunas: “person.id” e “person.name”. Ele ignora o “pessoa”. prefixo, e ignora os campos “ID”. Em seguida, todos os registros são anexados e atribuídos novos ids. Ambas estas operações podem ser realizadas através da interface AppAdmin web.

### 7.27.2 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', 'wb') as dumpfile:
    db.export_to_csv_file(dumpfile)
```

Importar:

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

Ou em Python 3:

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.

### 7.27.3 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 uuid

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

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

db.person.uuid.default = db.thing.uuid.default = lambda: str(uuid.uuid4())

db.thing.owner_id.requires = IS_IN_DB(db, 'person.uuid', '%(name)s')

# usage example
if db(db.person).isempty():
    nid = str(uuid.uuid4())
    db.person.insert(uuid=nid, name='Massimo')
    db.thing.insert(name='Chair', owner_id=nid)
```

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 recebe um UUID único, mesmo que vários registros são inseridos em uma única transação.

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

```
def export():
    s = StringIO.StringIO()
    db.export_to_csv_file(s)
    response.headers['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 yat1.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.

#### 7.27.4 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>
```

Se você precisar serializar as linhas em qualquer outro formato XML com tags personalizadas, você pode facilmente fazer isso usando o universal ``ajudante TAG`` (descrito em *Capítulo 11 <# capítulo 08 # TAGs>* \_\_ ea sintaxe Python ``\* <iterable>`` permitida em chamadas de função:

```
>>> 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>
```

### 7.27.5 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

## 7.28 Selects com cache

O método de seleção também leva um argumento `cache`, cujo padrão é `None`. Para fins de armazenamento em cache, deve ser definido como um tuplo em que o primeiro elemento é o modelo do cache (`cache.ram`, `cache.disk`, etc), e o segundo elemento é o tempo de validade em segundo.

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.ram, 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.ram, 3600), cacheable=True)
```

## 7.29 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 optámos 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.

## 7.30 Características avançadas

### 7.30.1 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:

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

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

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.

### 7.30.2 ``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 por exemplo que você deseja armazenar uma estrutura serializado dados Python em um campo no formato JSON. Aqui está como isso poderia ser feito:

```
>>> 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}]
```

Outra maneira de fazer a mesma é usando um campo do tipo ``SQLCustomType``, como discutido na próxima *personalizado tipos Field` <#Custom\_Field\_Types>`* seção.

### 7.30.3 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.

Isto é melhor explicado através de alguns exemplos.

```
>>> db.define_table('person', Field('name'))
<Table person (id, name)>
>>> def pprint(callback, *args):
...     print "%s%s" % (callback, args)
...
>>> db.person._before_insert.append(lambda f: pprint('before_insert', f))
>>> db.person._after_insert.append(lambda f, i: pprint('after_insert', f, i))
>>> db.person.insert(name='John')
before_insert(<OpRow {'name': 'John'}>,)
after_insert(<OpRow {'name': 'John'}>, 1L)
1L
>>> db.person._before_update.append(lambda s, f: pprint('before_update', s, f))
>>> db.person._after_update.append(lambda s, f: pprint('after_update', s, f))
>>> db(db.person.id == 1).update(name='Tim')
before_update(<Set ("person"."id" = 1)>, <OpRow {'name': 'Tim'}>)
after_update(<Set ("person"."id" = 1)>, <OpRow {'name': 'Tim'}>)
1
>>> db.person._before_delete.append(lambda s: pprint('before_delete', s))
>>> db.person._after_delete.append(lambda s: pprint('after_delete', s))
>>> db(db.person.id == 1).delete()
before_delete(<Set ("person"."id" = 1)>,)
after_delete(<Set ("person"."id" = 1)>,)
1
```

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

Esquema de banco de dados pode definir relacionamentos que de disparam exclusão de registros relacionados, conhecidos como cascade. A DAL não é informado quando um registro é excluído devido a um cascade. Portanto, não o callback *\*\_delete* nunca vai ser chamado como consequência de uma exclusão em cascata.

### 7.30.4 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.

``Common\_filter`` s será discutido nos próximos *filtros em comum* <#common\_filters> seção \_\_.

### 7.30.5 campos comuns e multi-tenancy

``Db.\_common\_fields`` é uma lista de campos que devem pertencem a todas as tabelas. Esta lista também pode conter tabelas e entende-se que todos os campos da tabela.

Por exemplo, ocasionalmente, você se encontra em necessidade de adicionar uma assinatura a todas as suas tabelas, mas os ``tabelas Auth``. Neste case, depois de ``auth.define\_tables()`` , mas antes de definir qualquer outra tabela, inserir:

```
db._common_fields.append(auth.signature)
```

Um campo é especial: ``request\_tenant``, você pode definir um nome diferente em ``db.\_request\_tenant``. Este campo não existe, mas você pode criá-lo e adicioná-lo a qualquer um dos seus quadros (ou todas elas):

```
db._common_fields.append(Field('request_tenant',
                               default=request.env.http_host,
                               writable=False))
```

Para cada mesa com tal campo um, todos os registros para todas as consultas são sempre filtrados automaticamente por:

```
db.table.request_tenant == db.table.request_tenant.default
```

e para cada registro inserido, este campo é definido como o valor padrão. No exemplo acima, nós escolhemos:

```
default = request.env.http_host
```

Isso significa que temos escolhido para perguntar nosso aplicativo para filtrar todas as tabelas em todas as consultas com:

```
db.table.request_tenant == request.env.http_host
```

Este truque simples nos permitem transformar qualquer aplicativo em um aplicativo multi-tenant. Mesmo que executar uma instância do aplicativo e usar um único banco de dados, quando o aplicativo é acessado em dois ou mais domínios os visitantes poderão ver dados diferentes dependendo do domínio (no exemplo, o nome de domínio é recuperado do ``pedido.env.http\_host``).

Você pode desativar filtros multi-tenancy usando ``ignore\_common\_filters = True`` em ``tempo de criação Set``:

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

### 7.30.6 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.

### 7.30.7 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 gluon.dal import SQLCustomType
>>> ipv4 = SQLCustomType(type='string', native='integer',
...                      encoder=lambda x : str(ip2int(x)), decoder=int2ip)
>>> db.define_table('website',
...                 Field('name'),
...                 Field('ipaddr', type=ipv4))
<Table website (id, name, ipaddr)>
>>> db.website.insert(name='wikipedia', ipaddr='91.198.174.192')
1
>>> db.website.insert(name='google', ipaddr='172.217.11.174')
2
>>> db.website.insert(name='youtube', ipaddr='74.125.65.91')
3
>>> db.website.insert(name='github', ipaddr='207.97.227.239')
4
>>> rows = db(db.website.ipaddr > '100.0.0.0').select(orderby=~db.website.ipaddr)
>>> for row in rows:
...     print row.name, row.ipaddr
...
github 207.97.227.239
google 172.217.11.174
```

``SQLCustomType`` 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.



Esta característica é marcado como experimental. Na prática, tem sido no py4web por um longo tempo e ele funciona, mas ele pode fazer o código não é portátil, por exemplo, quando o tipo nativo é específico do banco de dados.

Ele não funciona no Google App Engine NoSQL.

### 7.30.8 Usando DAL sem definir tabelas

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

```
from gluon import DAL
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 gluon import DAL
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.

### 7.30.9 PostGIS, SpatiaLite, e MS Geo (experimental)

Os suportes DAL APIs geográficas usando PostGIS (para PostgreSQL), SpatiaLite (para SQLite), e MSSQL e extensões espaciais. Este é um recurso que foi patrocinado pelo projeto Sahana e implementado por Denes Lengyel.

DAL fornece geometria e geografia campos tipos e as seguintes funções:

```
st_asgeojson (PostGIS only)
st_astext
st_contains
st_distance
st_equals
st_intersects
st_overlaps
st_simplify (PostGIS only)
st_touches
st_within
st_x
st_y
```

aqui estão alguns exemplos:

```
>>> from gluon.dal import DAL, Field, geoPoint, geoLine, geoPolygon
>>> db = DAL("mssql://user:pass@host/db")
>>> sp = db.define_table('spatial', Field('loc', 'geometry()'))
```

A seguir, insira um ponto, uma linha, e um polígono:

```
>>> sp.insert(loc=geoPoint(1, 1))
1
```

```
>>> sp.insert(loc=geoLine((100, 100), (20, 180), (180, 180)))
2
>>> sp.insert(loc=geoPolygon((0, 0), (150, 0), (150, 150), (0, 150), (0, 0)))
3
```

Notar que

```
rows = db(sp).select()
```

Sempre retorna os dados de geometria serializados como texto. Você também pode fazer o mesmo mais explicitamente usando ``ST\_AsText()``:

```
>>> print db(sp).select(sp.id, sp.loc.st_astext())
spatial.id, spatial.loc.STAsText()
1, "POINT (1 2)"
2, "LINESTRING (100 100, 20 180, 180 180)"
3, "POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))"
```

Você pode pedir a representação nativa usando ``st\_asgeojson()`` (em apenas PostGIS):

```
>>> print db(sp).select(sp.id, sp.loc.st_asgeojson().with_alias('loc'))
spatial.id, loc
1, [1, 2]
2, [[100, 100], [20 180], [180, 180]]
3, [[[0, 0], [150, 0], [150, 150], [0, 150], [0, 0]]]
```

(Nota uma matriz é um ponto, uma matriz de matrizes é uma linha, e um conjunto de matriz de matrizes é um polígono).

Aqui estão exemplo de como usar as funções geográficas:

```
>>> query = sp.loc.st_intersects(geoLine((20, 120), (60, 160)))
>>> query = sp.loc.st_overlaps(geoPolygon((1, 1), (11, 1), (11, 11), (11, 1), (1, 1)))
>>> query = sp.loc.st_contains(geoPoint(1, 1))
>>> print db(query).select(sp.id, sp.loc)
spatial.id, spatial.loc
3, "POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))"
```

distâncias calculadas também pode ser recuperado como números de ponto flutuante:

```
>>> dist = sp.loc.st_distance(geoPoint(-1, 2)).with_alias('dist')
>>> print db(sp).select(sp.id, dist)
spatial.id, dist
1, 2.0
2, 140.714249456
3, 1.0
```

### 7.30.10 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.

PY4WEB fornece um script que faz este trabalho para você:

```
cd py4web
python scripts/cpdb.py \
  -f applications/app/databases \
  -y 'sqlite://storage.sqlite' \
  -Y 'postgres://username:password@localhost/mydb' \
  -d ../gluon
```

Depois de executar o script, você pode simplesmente mudar a sequência de conexão no modelo e tudo deve funcionar fora da caixa. Os novos dados devem estar lá.

Este script fornece várias opções de linha de comando que permite que você mover dados de uma aplicação para outra, mover todas as tabelas ou apenas algumas mesas, limpar os dados nas tabelas. Para mais informações tentativa:

```
python scripts/cpdb.py -h
```

### 7.30.11 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.

O arquivo “glúons / dal.py” define, entre outras, as seguintes 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.

“Glúon / dal.py” no momento de escrever este livro, contém os seguintes adaptadores:

```
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://...')
```

o prefixo na strings de uri define o adaptador. O mapeamento é definido no seguinte dicionário também em “gluão / dal.py”:

```

ADAPTERS = {
    'sqlite': SQLiteAdapter,
    'spatialite': SpatialiteAdapter,
    'sqlite:memory': SQLiteAdapter,
    'spatialite:memory': SpatialiteAdapter,
    'mysql': MySQLAdapter,
    'postgres': PostgreSQLAdapter,
    'postgres:psycopg2': PostgreSQLAdapter,
    'postgres2:psycopg2': NewPostgreSQLAdapter,
    'oracle': OracleAdapter,
    'mssql': MSSQLAdapter,
    'mssql2': MSSQL2Adapter,
    'mssql3': MSSQL3Adapter,
    'mssql4': MSSQL4Adapter,
    'vertica': VerticaAdapter,
    'sybase': SybaseAdapter,
    'db2': DB2Adapter,
    'teradata': TeradataAdapter,
    'informix': InformixAdapter,
    'informix-se': InformixSEAdapter,
    'firebird': FireBirdAdapter,
    'firebird_embedded': FireBirdAdapter,
    'ingres': IngresAdapter,
    'ingresu': IngresUnicodeAdapter,
    'sapdb': SAPDBAdapter,
    'cubrid': CubridAdapter,
    'jdbc:sqlite': JDBCSQLiteAdapter,
    'jdbc:sqlite:memory': JDBCSQLiteAdapter,
    'jdbc:postgres': JDBCPostgreSQLAdapter,
    'gae': GoogleDatastoreAdapter, # discouraged, for backward compatibility
    'google:datastore': GoogleDatastoreAdapter,
    'google:datastore+ndb': GoogleDatastoreAdapter,
    'google:sql': GoogleSQLAdapter,
    'couchdb': CouchDBAdapter,
    'mongodb': MongoDBAdapter,

```

```
'imap': IMAPAdapter
}
```

a string URI é então analisado com mais detalhes pelo próprio adaptador.

Para qualquer adaptador que você pode substituir o motorista com um diferente:

```
import MySQLdb as mysqldb
from gluon.dal import MySQLAdapter
MySQLAdapter.driver = mysqldb
```

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

```
db =DAL(..., driver_args={}, adapter_args={})
```

## 7.31 Pegadinhas

### 7.31.1 SQLite

SQLite não apoiar caindo e alterar colunas. Isso significa que as migrações py4web irão trabalhar até certo ponto. Se você excluir um campo de uma tabela, a coluna permanecerá no banco de dados, mas será invisível para py4web. Se você decidir para restabelecer a coluna, py4web vai tentar recriá-la e falhar. Neste case, você deve definir ``fake\_migrate = True`` de modo que os metadados é reconstruído sem tentar adicionar a coluna novamente. Além disso, pela mesma razão, **SQLite** não tem conhecimento de qualquer mudança de tipo de coluna. Se você inserir um número em um campo string, ele será armazenado como string. Se posteriormente você alterar o modelo e substituir o tipo “string” com o tipo “inteiro”, SQLite continuará a manter o número como uma string e isso pode causar problemas quando você tenta extrair os dados.

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.

### 7.31.2 MySQL

O MySQL não suporta múltiplos ALTER TABLE em uma única transação. Isto significa que qualquer processo de migração é quebrado em vários commits. Se algo acontece que faz com que uma falha é possível quebrar uma migração (os metadados py4web não estão mais em sincronia com a estrutura da tabela real no banco de dados). Isto é lamentável, mas ela pode ser prevenida (migre uma mesa no tempo) ou pode ser fixado a posteriori (reverter o modelo py4web ao que corresponde à estrutura da tabela na banco de dados, conjunto ``fake\_migrate = True`` e depois os metadados foi reconstruído, conjunto ``fake\_migrate = False`` e migrar a tabela de novo).

### 7.31.3 Google SQL

Google SQL tem os mesmos problemas que o MySQL e muito mais. Em particular metadados da tabela em si deve ser armazenado no banco de dados em uma tabela que não é migrado por py4web. Isso ocorre porque o Google App Engine tem um sistema de arquivos somente leitura. migrações PY4WEB em Google SQL combinadas com a questão MySQL descrito acima pode resultar em corrupção de metadados. Novamente, isso pode ser evitado (através da migração da mesa de uma vez e, em seguida, definindo migrar = False para que a tabela de metadados não é acessado mais) ou pode fixa a posteriori (acessando o banco de dados usando o painel do Google e excluir qualquer entrada corrompido da mesa chamado ``

py4web\_filesystem`.

### 7.31.4 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 ação”. 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)
```

### 7.31.5 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.

### 7.31.6 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 sequenciais. 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.





## A RESTAPI

Desde a versão 19.5.10 PyDAL inclui uma API RESTful chamado RestAPI. É inspirado por GraphQL mas não é bem a mesma coisa, porque é menos poderoso, mas, no espírito do web2py, mais prático e mais fácil de usar. Como GraphQL RestAPI permite que um cliente para consulta de informações usando o método GET e permite especificar alguns detalhes sobre o formato da resposta (que referências a seguir, e como desnormalizar os dados). Ao contrário GraphQL ele permite que o servidor para especificar uma política e restringir quais consultas são permitidas e quais não são. Eles podem ser avaliados de forma dinâmica por solicitação com base no usuário e o estado do servidor. Como o nome implica RestAPI permite que todos os métodos standard GET, POST, PUT e DELETE. Cada um deles pode ser ativado ou desativado com base na política, para tabelas individuais e campos individuais.

Nos exemplos abaixo assumimos um aplicativo chamado “super-heróis” e o seguinte modelo:

```
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'))
```

Também assumimos o seguinte controlador ``rest.py``:

```
from pydal.dbapi import RestAPI, Policy

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

# 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'])
def api(tablename, rec_id=None):
    return RestAPI(db, policy)(request.method,
                                tablename,
                                rec_id,
                                request.GET,
                                request.POST
                                )

```

A política é por tabela (ou \* para todas as tabelas e por método. Autorizar pode ser verdade (permitir), False (negar) ou uma função com a assinatura (método, tablename, record\_id, get\_vars, post\_vars) que retorna Verdadeiro / Falso . para a política GET pode especificar uma lista de padrões de consulta permitidos (\* para todos). um padrão de consulta será comparado com as chaves na cadeia de consulta.

A acção acima referida é exposto como:

```
/superheroes/rest/api/{tablename}
```

**\*\* Sobre request.POST \*\*:** Manter em mente que **\*\* request.POST** contém apenas os dados do formulário que é publicado utilizando um formulário HTML normal **\*\*\*** ou javascript **\*\* FormData \*\*** objeto. Se você postar apenas objeto simples (por exemplo, `axios.post ( "path / to / api", {campo: "alguns"})` **\*\***) você deve passar request.json **\*\*** em vez de request.POST, desde Este último irá conter request-corpo apenas cru que é corda, não json. Consulte a documentação bottle.py para mais detalhes.

## 8.1 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`.

`{Algo}` pode ser o nome de um campo na tabela foi consultado como em:

**\*\* Todos os super-heróis chamado de "Superman" \*\***

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

Pode ser um nome de um campo de uma tabela referida pela tabela foi consultado como em:

**\*\* Todos os super-heróis com a identidade real "Clark Kent" \*\***

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

Pode ser o nome de um campo de uma tabela que se refere ao neen tabela consultada como em:

**\*\* Todos os super-heróis com qualquer superpotência tag com força > 90 \*\***

```
/superheroes/rest/api/superhero?superhero.tag.strength.gt=90
```

(Aqui tag é o nome da tabela de ligação, o anterior `superhero` é o nome do campo que faz referência ao quadro seleccionado e `strength` é o nome do campo utilizado para filtro)

Ele também pode ser um campo da tabela referenciada por uma tabela de muitos-para-muitos ligado como em:

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

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

A chave para entender a sintaxe acima é para quebrá-lo da seguinte forma:

```
superhero?superhero.tag.superpower.description.eq=Flight
```

e lê-lo como:

selecionar registros da tabela **super-herói** referido pelo campo **super-herói** da tabela **tag** quando a superpotência campo dos ditos pontos de mesa para um recorde com a descrição **eq** ual para “Flight”.

A consulta permite modificadores adicionais, por exemplo

```
@offset=10
@limit=10
@order=name
@model=true
@lookup=real_identity
```

Os 3 primeiros são óbvias. @model retorna uma descrição JSON do modelo de banco de dados. Lookup desnormaliza o campo vinculado.

Aqui estão alguns exemplos práticos:

URL:

```
/superheroes/rest/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/rest/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": "2019-05-19T05:38:00.098292",
  "model": [
    {
      "regex": "[1-9]\\d*",
      "name": "id",
      "default": null,
      "required": false,
      "label": "Id",
      "post_writable": true,
      "referenced_by": [],
      "unique": false,
      "type": "id",
      "options": null,
      "put_writable": true
    },
    {
      "regex": null,
      "name": "name",
      "default": null,
      "required": false,
      "label": "Name",
      "post_writable": true,
      "unique": false,
      "type": "string",
      "options": null,
      "put_writable": true
    },
    {
      "regex": null,
      "name": "real_identity",
      "default": null,
      "required": false,
      "label": "Real Identity",
      "post_writable": true,
      "references": "person",
      "unique": false,
```

```

        "type": "reference",
        "options": null,
        "put_writable": true
    },
    "api_version": "0.1"
}

```

URL:

```
/superheroes/rest/api/superhero?@lookup=real_identity
```

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/rest/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/rest/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": "2019-05-19T05:38:00.192180",
"api_version": "0.1"
}

```

URL:

```
/superheroes/rest/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/rest/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 (que é uma linha única, dividida para facilitar a leitura):

```

/superheroes/rest/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 (que é uma linha única, dividida para facilitar a leitura):

```
/superheroes/rest/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
},

```

```
        "id": 3
    }
],
"timestamp": "2019-05-19T05:38:00.355181",
"api_version": "0.1"
}
```

URL (que é uma linha única, dividida para facilitar a leitura):

```
/superheroes/rest/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": "2019-05-19T05:38:00.396583",
"api_version": "0.1"
}

```

URL:

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

Note todas resposta RestAPI ter os campos



```
{
  "api_version": ...
  "timestamp": ...
  "status": ...
  "code": ...
}
```

e alguns campos opcionais:

```
{
  "count": ... (total matching, not total returned, for GET)
  "items": ... (in response to a GET)
  "errors": ... (usually validation error0
  "models": ... (usually if status != success)
  "message": ... (is if error)
}
```

As especificações exatas estão sujeitos a mudança uma vez que este é um novo recurso.



---

## Linguagem de template YATL

---

py4web usa Python para os seus templates, controladores e pontos de vista, embora ele usa uma sintaxe Python ligeiramente modificada nas vistas para permitir que o código mais legível, sem impor quaisquer restrições sobre o uso adequado Python.

usos py4web ``[...]`` para escapar código Python embutido em HTML. A vantagem de usar colchetes em vez de colchetes é que é transparente para todos os editores comum HTML. Isso permite que o desenvolvedor usar esses editores para criar visualizações py4web.

Desde o desenvolvedor está incorporação de código Python em HTML, o documento deve ser recuado de acordo com as regras HTML, e não regras Python. Portanto, permitimos que sem recuo Python dentro das ``[...]`` tags. Desde Python normalmente usa recuo para delimitar blocos de código, precisamos de uma maneira diferente para delimitá-los; é por isso que as marcas de linguagem template py4web usar da palavra-chave Python ``pass``.

Um bloco de código é iniciado com uma linha que termina com dois pontos e as extremidades com uma linha que se inicia com ``pass``. A palavra-chave ``pass`` não é necessário quando o fim do bloco é óbvio a partir do contexto.

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<br />[[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<br />""", escape=False)
response.write("""</body></html>""", escape=False)
```

`` Response.write`` escreve para o `` response.body``.

Quando há um erro em uma visão py4web, o relatório de erro mostra o código gerado vista, e não o ponto de vista real como escrito pelo desenvolvedor. Isso ajuda o desenvolvedor de depuração do código, destacando o código real que é executado (que é algo que pode ser depurado com um editor de HTML ou o inspetor DOM do navegador).

Observe também que:

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

gera

```
response.write(x)
```

Variáveis injetadas no HTML desta forma são escapadas por padrão. O escape é ignorado se `` x`` é um objeto `` XML``, mesmo que escape é definida como `` True``.

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, escapados 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 escapados, 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.

Isso tudo é feito de forma transparente. Você nunca precisa (e não deve) chamar o método *response.write* explicitamente.

## 9.1 Sintaxe básica

O linguagem de template py4web suporta todas as estruturas de controle Python. Aqui nós fornecemos alguns exemplos de cada um deles. Eles podem ser aninhados de acordo com a prática de programação habitual.

### 9.1.1 `` Para ... in``

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

```
[[items = ['a', 'b', 'c']]]
<ul>
[[for item in items:]]<li>[[=item]]</li>[[pass]]
```

```
</ul>
```

que produz:

```
<ul>
<li>a</li>
<li>b</li>
<li>c</li>
</ul>
```

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]]
<ul>
[[while k > 0:]]<li>[[k]][[k = k - 1]]</li>[[pass]]
</ul>
```

que produz:

```
<ul>
<li>3</li>
<li>2</li>
<li>1</li>
</ul>
```

### 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``

Também é possível usar ``tentar ... declarações except`` nas vistas com uma ressalva. Considere o seguinte exemplo:

```
[[try:]]
Hello [[= 1 / 0]]
[[except:]]
division by zero
[[else:]]
no division by zero
[[finally:]]
<br />
[[pass]]
```

Ela irá produzir o seguinte resultado:

```
Hello division by zero
<br />
```

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))]]
<ul>
[[=itemize1('www.google.com')]]
</ul>
```

produz o seguinte resultado:

```
<ul>
<li><a href="http://www.google.com">www.google.com</a></li>
</ul>
```

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):]]  
<li><a href="http://[[link]]">[[link]]</a></li>  
[[return]]  
<ul>  
[[itemize2('www.google.com')]]  
</ul>
```

Ela produz exatamente 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.

Há uma ressalva: funções definidas dentro de uma visão deve terminar com uma declaração ``return``, ou o recuo automático falhará.





## Helpers YATL

Considere o seguinte código em um ponto de vista:

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

ele é processado como:

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

``Div`` é uma classe auxiliar, ou seja, algo que pode ser usado para construir HTML programaticamente. Corresponde ao HTML ``<div> tag``.

Argumentos posicionais são interpretados como objetos contidos entre as tags de abrir e fechar. argumentos nomeados que começam com um sublinhado são interpretados como atributos de tags HTML (sem o sublinhado). Alguns helpers também têm argumentos nomeados que não começam com sublinhado; estes argumentos são específicos do tag.

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 a mesma saída que antes).

O seguinte conjunto de helpers:

```
`` A``, `` BEAUTIFY``, `` BODY``, `` CAT``, `` CODE``, `` div``, `` EM``, `` Form``, `` H1``, `` h2``, `` H3``, ``
H4``, `` H5``, `` H6``, `` HEAD``, `` HTML``, `` I``, `` IMG``, `` INPUT``, `` label``, `` LI``, `` LINK``, `` Meta``, `` METATAG``,
OL``, `` OPTION``, `` PRE``, `` SELECT``, `` SPAN``, `` STRONG``, `` TABLE``, `` TAG``, `` TBODY``, ``
TD``, `` TEXTAREA``, `` TH``, `` THEAD``, `` TR``, `` UL``, `` XML``, `` sanitize``, `` xmlescape``
```

pode ser usado para construir expressões complexas que podem então ser serializado para XML. Por exemplo:

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

é prestado:

```
<div class="myclass"><b><i>hello <lt;world></i></b></div>
```

Ajudantes também podem ser serializados em cordas, de forma equivalente, com a ``\_\_str\_\_`` e os ``métodos xml``:

```
>>> print str(DIV("hello world"))
<div>hello world</div>
>>> print DIV("hello world").xml()
<div>hello world</div>
```

O mecanismo de helpers em py4web é mais do que um sistema para gerar HTML sem concatenar strings. Ele fornece uma representação do lado do servidor do objecto do modelo de documento (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><span>ab</span>c</div>
>>> del a[1]
>>> a.append(B('x'))
>>> a[0][0] = 'y'
>>> print a
<div><span>yb</span><b>x</b></div>
```

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"><span class="t">ab</span>c</div>
```

Note, o conjunto completo de componentes podem ser acessados através de uma lista de chamada ``a.components``, eo conjunto completo de atributos podem ser acessados através de um dicionário chamado ``a.attributes``. Assim, ``a[i]`` é equivalente a ``a.components[i]`` quando i é um número inteiro, e um ``[s]`` é equivalente a ``a.attributes[s]`` quando s é uma cadeia de caracteres.

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, hífen) e, portanto, não podem ser usados como nomes de argumentos de palavra-chave. Por exemplo:

```
DIV('text', _data-role='collapsible')
```

não vai funcionar porque `<quotechar> _data-papel <quotechar>` inclui um hífen, que irá produzir um erro de sintaxe Python.

Nesses casos, você tem um par de opções. É possível utilizar a ``data`` argumento (desta vez sem um sublinhado) para passar um dicionário de atributos relacionados sem a sua hífen líder, e a saída terá as combinações desejadas, por exemplo

```
>>> print DIV('text', data={'role': 'collapsible'})
<div data-role="collapsible">text</div>
```

ou você pode em vez disso passar os atributos como um dicionário e fazer uso de ``\*\*`` notação ``argumentos de função do Python, que mapeia um dicionário de (key: value) pares em um conjunto de argumentos de palavra-chave:

```
>>> print DIV('text', **{'_data-role': 'collapsible'})
<div data-role="collapsible">text</div>
```

Note-se que as entradas mais elaboradas irá introduzir entidades de caracteres HTML, mas eles vão trabalhar, no entanto, por exemplo,

```
>>> print DIV('text', data={'options': '{"mode": "calbox", "useNewStyle": true}'})
<div data-options="{&quot;mode&quot;;&quot;calbox&quot;;&quot;useNewStyle&quot;;true}">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>
```

## 10.1 ``XML``

``XML`` é um objeto usado para encapsular texto que não deve ser escapado. O texto pode ou não conter XML válido. Por exemplo, poderia conter JavaScript.

O texto neste exemplo é escapado:

```
>>> print DIV("<b>hello</b>")
<div>&lt;b&gt;hello&lt;/b&gt;</div>
```

usando ``XML`` você pode impedir escapar:

```
>>> print DIV(XML("<b>hello</b>"))
<div><b>hello</b></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-escapou de entrada executável como este (por exemplo, entrou no corpo de um comentário em um blog) não é seguro, porque pode ser usado para gerar ataques Cross Site Scripting (XSS) contra outros visitantes da página.

O py4web ``helper XML`` pode higienizar nosso texto para evitar injeções e escapar todas as tags exceto aqueles que você permitir explicitamente. Aqui está um exemplo:

```
>>> print XML('<script>alert("unsafe!")</script>', sanitize=True)
&lt;script&gt;alert(&quot;unsafe!&quot;)&lt;/script&gt;
```

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/'],
    allowed_attributes={'a': ['href', 'title'],
        'img': ['src', 'alt'], 'blockquote': ['type']})
```

## 10.2 Built-in helpers

### 10.2.1 ``A``

Este assistente é usado para construir ligações.

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

### 10.2.2 ``BODY``

Este assistente faz com que o corpo de uma página.

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

### 10.2.3 ``CAT``

Este assistente concatena outros helpers, mesmo como TAG [ "" ].

```
>>> print CAT('Here is a ', A('link', _href=URL()), ', and here is some ', B('bold
text'), '.')
Here is a <a href="/app/default/index">link</a>, and here is some <b>bold text</b>.
```

### 10.2.4 ``CODE``

Este assistente executa da sintaxe para o Python, C, C ++, o código HTML e py4web, e é preferível a ``PRE`` para listagens de código. ``CODE`` também tem a capacidade de criar links para a documentação py4web API.

Aqui está um exemplo de destacar seções de código Python.

```
>>> print CODE('print "hello"', language='python').xml()
```

```
<table><tr style="vertical-align:top;">
  <td style="min-width:40px; text-align: right;"><pre style="
    font-size: 11px;
    font-family: Bitstream Vera Sans Mono,monospace;
    background-color: transparent;
    margin: 0;
    padding: 5px;
    border: none;
    color: #A0A0A0;
">1.</pre></td><td><pre style="
  font-size: 11px;
  font-family: Bitstream Vera Sans Mono,monospace;
  background-color: transparent;
  margin: 0;
  padding: 5px;
  border: none;
  overflow: auto;
  white-space: pre !important;
```

```
"><span style="color:#185369; font-weight: bold">print </span>
<span style="color: #FF9966">"hello"</span></pre></td></tr></table>
```

Aqui está um exemplo semelhante para HTML

```
>>> print CODE(' <html><body>[=request.env.remote_add]</body></html>',
...             language='html')
```

```
<table>...<code>...
<html><body>[=request.env.remote_add]</body></html>
...</code>...</table>
```

Estes são os argumentos padrão para o ``CODE`` helper:

```
CODE("print 'hello world'", language='python', link=None, counter=1, styles={})
```

Os valores suportados para o argumento ``language`` são “python”, “html\_plain”, “c”, “cpp”, “py4web”, e “html”. Os “HTML” interpreta linguagem tags como código “py4web”, enquanto “html\_plain” não.

Se um ``valor link`` é especificado, por exemplo “/ examples / global / vars /”, referências py4web API no código estão ligados a documentação no link URL. Por exemplo “pedido” estaria ligada à “/ examples / / global vars / request”. No exemplo acima, o URL de ligação é tratada pelo “vars” acção na “global.py” controlador que é distribuído como parte do py4web “exemplos” aplicação.

O argumento ``counter`` é utilizado para a numeração de linha. Ele pode ser configurado para qualquer um dos três valores diferentes. Pode ser ``None`` para não números de linha, um valor numérico especificando o número inicial, ou uma string. Se o contador está definido para uma cadeia, ele é interpretado como um aviso, e não há números de linha.

O argumento styles`` é um pouco complicado. Se você olhar para o HTML gerado acima, que contém uma tabela com duas colunas, e cada coluna tem um estilo próprio declarou em linha usando CSS. Os `` atributos styles`` permite substituir esses dois estilos CSS. Por exemplo:

```
CODE(..., styles={'CODE':'margin: 0;padding: 5px;border: none;'})
```

O atributo ``styles`` deve ser um dicionário, e permite que duas chaves possíveis: ``CODE`` para o estilo do código atual, e ``LINENUMBERS`` para o estilo da coluna esquerda, que contém a linha números. Lembre-se que estes estilos substituir completamente os estilos padrão e não são simplesmente adicionados a eles.

### 10.2.5 ``Div``

Todos os helpers além de ``XML`` são derivados de ``div`` e herdar seus métodos básicos.

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

### 10.2.6 ``EM``

Insiste no seu conteúdo.

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

### 10.2.7 ``Form``

Este é um dos mais helpers importantes. Na sua forma mais simples, ele só faz um `` <form> ... </ form>

tag `<<`, mas porque helpers são objetos e ter conhecimento do que eles contêm, eles podem processar formulários enviados (por exemplo, executar a validação de os campos). Isso será discutido em detalhes no *Capítulo 12* <# capítulo-12> `<<`.

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

O “enctype” é “multipart / form-data” por padrão.

O construtor de um `<< Form<<`, e de `<< SQLFORM<<`, também pode tomar um argumento especial chamado `<< hidden<<`. Quando um dicionário é passado como `<< hidden<<`, seus itens são convertidos em campos de entrada “escondido”. Por exemplo:

```
>>> print FORM(hidden=dict(a='b'))
<form enctype="multipart/form-data" action="" method="post">
<input value="b" type="hidden" name="a" /></form>
```

## 10.2.8 `<< H1<<`, `<< h2<<`, `<< H3<<`, `<< H4<<`, `<< H5<<`, `<< H6<<`

Estes helpers são para títulos dos parágrafos e subtítulos:

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

## 10.2.9 `<< HEAD<<`

Para marcar a cabeça de uma página HTML.

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

## 10.2.10 `<< HTML<<`

Este assistente é um pouco diferente. Além de fazer os `<< <html> <<` tag, que precederá a tag com um fio doctype.

```
>>> print HTML(BODY('<hello>', XML('<b>world</b>')))
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html><body>&lt;hello&gt;<b>world</b></body></html>
```

O auxiliar HTML também leva alguns argumentos opcionais adicionais que têm o seguinte padrão:

```
HTML(..., lang='en', doctype='transitional')
```

onde DOCTYPE pode ser ‘rigorosa’, ‘transitório’, ‘do conjunto de quadros’, ‘html5’, ou uma corda tipo de documento completo.

## 10.2.11 `<< I<<`

Este assistente torna o seu conteúdo em itálico.

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

## 10.2.12 ``IMG``

Ele pode ser usado para imagens incorporar em HTML:

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

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'))
...
<a href="/myapp/default/index">
{ alt="My Logo" }
</a>
```

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

Ele também leva um argumento especial opcional chamado “valor”, distinto <code>\_value</code>. Este último define o valor padrão para o campo de entrada; o ex define seu valor atual. Para uma entrada do tipo “text” <code>\_type</code>, os antigos substituí o último:

```
>>> print INPUT(_name='test', _value='a', value='b')
<input value="b" name="test" />
```

Para os botões de rádio, ``INPUT`` define seletivamente o “marcada” atributo:

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

e similarmente para caixas de seleção:

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

## 10.2.14 ``Label``

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

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

### 10.2.15 ``LI``

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

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

### 10.2.16 ``OL``

Fica para a lista ordenada. A lista deve conter tags LI. `` Argumentos OL`` que não são `` objetos LI`` são automaticamente fechados em `` <li> ... </li> `` tags.

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

### 10.2.17 ``OPTION``

Isso só deve ser usado como parte de um `` / `` combinação OPTION`` SELECT``.

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

Como no caso de `` INPUT``, py4web fazer uma distinção entre <quotechar> \_value <quotechar> (o valor da opção), e “valor” (o valor atual do delimitador selecionar). Se forem iguais, a opção é “selecionado”.

```
>>> print SELECT('a', 'b', value='b'):
<select>
<option value="a">a</option>
<option value="b" selected="selected">b</option>
</select>
```

### 10.2.18 ``P``

Isto é para marcar um parágrafo.

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

### 10.2.20 ``SCRIPT``

Esta é incluir ou vincular um script, como JavaScript. O conteúdo entre as tags é processado como um comentário HTML, para o benefício dos navegadores realmente antigos.

```
>>> print SCRIPT('alert("hello world");', _type='text/javascript')
<script type="text/javascript"><!--
alert("hello world");
```



```
//--></script>
```

### 10.2.21 ``SELECT``

Faz um ``<selecionar> ... </select>`` tag . Isto é usado com o `` helper OPTION. Esses `` argumentos SELECT`` que não são `` objetos OPTION`` são automaticamente convertidas em opções.

```
>>> print SELECT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<select id="0" class="test">
<option value="&lt;hello&gt;">&lt;hello&gt;</option>
<option value="&lt;b&gt;world&lt;/b&gt;"><b>world</b></option>
</select>
```

### 10.2.22 ``SPAN``

Semelhante a ``div`` mas utilizado para marcação em linha (em vez de bloco) conteúdo.

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

### 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`` espera conteúdo TD``; argumentos que não são ``objetos TD`` são convertidos automaticamente.

```
>>> print TABLE(TR('a', 'b'), TR('c', 'd'))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>
```

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

### 10.2.26 ``TEXTAREA``

Este assistente faz uma `<textarea> ... </textarea>` tag ``.

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

A única ressalva é que o seu “valor” opcional substitui seu conteúdo (HTML interna)

```
>>> print TEXTAREA(value="<hello world>", _class="test")
<textarea class="test" cols="40" rows="10">&lt;hello world&gt;</textarea>
```

### 10.2.27 ``TH``

Este é utilizado em vez de ``TD`` em cabeçalhos de tabela.

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

### 10.2.30 ``TR``

Palavras chave uma linha da tabela. Ele deve ser processado dentro de uma tabela e conter ``<td> ... </td>`` tags. ``Argumentos TR`` que não são ``objetos TD`` serão convertidos automaticamente.

```
>>> print TR('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<tr id="0" class="test"><td>&lt;hello&gt;</td><td><b>world</b></td></tr>
```

### 10.2.31 ``TT``

Etiquetas de texto como máquina de escrever texto (monoespaçada).

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

### 10.2.32 ``UL``

Significa uma lista desordenada e deve conter ``itens LI``. Se o seu conteúdo não é marcado como ``LI``, ``UL`` faz isso automaticamente.

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

### 10.2.33 ``URL``

O helper URL é documentado em \*Capítulo 4 URL ../04\*

## 10.3 Helpers personalizados

### 10.3.1 ``TAG``

Às vezes você precisa para gerar tags XML personalizados. py4web fornece ``TAG``, um gerador de tag universal.

```
[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')]
```

Se o objeto TAG é criado com um nome vazio, ele pode ser usado para concatenar várias cadeias e helpers HTML juntos sem inseri-los em uma tag ao redor, mas este uso é obsoleto. Use o `` helper CAT`` vez.

Tags com auto-fechamento podem ser geradas com o helper TAG. O noma da tag deve terminar com um “/”.

```
[TAG['link/'](_href='http://py4web.com')]
```

gera o seguinte XML:

```
<link ref="http://py4web.com"/>
```

Note que ``TAG`` é um objecto, e ``TAG.name`` ou TAG [ “nome” ] `` é uma função que retorna uma classe auxiliar temporária.

### 10.3.2 ``MENU``

O auxiliar MENU leva uma lista de listas ou de tuplas da forma de ``response.menu`` e gera uma árvore-como estrutura usando listas não ordenadas representam o menu. Por exemplo:

```
>>> print MENU([[ 'One', False, 'link1'], [ 'Two', False, 'link2']])
<ul class="py4web-menu py4web-menu-vertical">
<li><a href="link1">One</a></li>
<li><a href="link2">Two</a></li>
</ul>
```

O primeiro item em cada lista / tupla é o texto a ser exibido para o item de menu dado.

O segundo produto em cada lista / tupla é um booleano que indica se o item de menu particular é activo (isto é, o produto actualmente seleccionada). Quando definido como verdadeiro, o ``MENU`` helper irá adicionar uma classe “py4web-menu ativo” para o ``<li>`` para esse item (você pode alterar o nome dessa classe, através do argumento “li\_active” a ``MENU``). Outra maneira de especificar o url ativo é passando-o directamente para ``MENU`` através do seu argumento “active\_url”.

O terceiro item em cada lista / tupla pode ser um HTML helper (que poderia incluir helpers aninhados), eo ``MENU`` helper irá simplesmente tornar esse helper em vez de criar seu próprio ``<a>`` tag.

Cada item do menu pode ter um quarto argumento que é um submenu aninhado (e assim por diante de forma recursiva):

```
>>> print MENU([[ 'One', False, 'link1', [[ 'Two', False, 'link2']] ]])
<ul class="py4web-menu py4web-menu-vertical">
<li class="py4web-menu-expand">
<a href="link1">One</a>
<ul class="py4web-menu-vertical">
<li><a href="link2">Two</a></li>
</ul>
</li>
</ul>
```

Um item de menu também pode ter um quinto elemento opcional, que é um boolean. Quando false, o item de menu é ignorado pelo auxiliar MENU.

O ``MENU`` helper leva os seguintes argumentos opcionais: - ``\_class``: o padrão é “menu py4web py4web-menu vertical” e define a classe dos elementos UL exteriores. - ``ul\_class``: o padrão é “py4web-menu vertical” e define a classe dos elementos UL internos. - ``li\_class``: o padrão é “py4web menu-expandir” e define a classe dos elementos LI internos. - ``li\_first``: permite adicionar uma classe para o primeiro elemento da lista. - ``li\_last``: permite adicionar uma classe para o último elemento da lista.

``MENU`` leva um argumento opcional ``mobile``. Quando ajustado para ``True`` em vez de construir um ``estrutura do menu UL`` recursiva ele retorna um ``suspensa SELECT`` com todas as opções de menu e um atributo ``onchange`` que redireciona para a página correspondente ao opção seleccionada. Isto é projetado uma representação de um menu alternativo que aumenta a usabilidade em pequenos dispositivos móveis, como telefones.

Normalmente, o menu é usado em um layout com a seguinte sintaxe:

```
[ [=MENU(response.menu, mobile=request.user_agent().is_mobile)]]
```

Desta forma, um dispositivo móvel é automaticamente detectado e o menu é processado em conformidade.

## 10.4 ``BEAUTIFY``

``BEAUTIFY`` é usado para representações de construção HTML de objetos compostos, incluindo listas, tuplas e dicionários:

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

retribuirá como:

```
<table>
<tr><td>a</td><td>:</td><td>hello<br />world</td></tr>
<tr><td>b</td><td>:</td><td>1<br />2</td></tr>
</table>
```

## 10.5 Server-side \* DOM \* e análise

### 10.5.1 ``elements``

O auxiliar DIV e todos os auxiliares derivados fornecer a métodos de pesquisa ``element`` e ``elements``.

``Retornos element`` o primeiro elemento filho correspondente a uma condição especificada (ou nenhum, se não fósforo).

``Elements`` retorna uma lista de todas as crianças correspondentes.

\*\* elemento \*\* e elementos \*\*\* usar a mesma sintaxe para especificar a condição de correspondência, que permite três possibilidades que podem ser misturados e combinados: jQuery-como expressões, jogo pelo valor do atributo exato, fósforo usando expressões regulares.

Aqui está um exemplo simples:

```
>>> a = DIV(DIV(DIV('a', _id='target', _class='abc')))
>>> d = a.elements('div#target')
>>> d[0][0] = 'changed'
>>> print a
<div><div><div id="target" class="abc">changed</div></div></div>
```

O argumento sem nome de ``elements`` é uma string, que pode conter: o nome de uma tag, o id de uma tag precedida por um símbolo de libra, a classe precedido por um ponto, o valor explícito de um atributo em parêntesis rectos.

Aqui estão 4 maneiras equivalentes para pesquisar a tag anterior id:

```
d = a.elements('#target')
d = a.elements('div#target')
d = a.elements('div[id=target]')
d = a.elements('div', _id='target')
```

Aqui estão 4 maneiras equivalentes para pesquisar a tag anterior por classe:

```
d = a.elements('.abc')
d = a.elements('div.abc')
d = a.elements('div[class=abc]')
d = a.elements('div', _class='abc')
```

Qualquer atributo pode ser usado para localizar um elemento (e não apenas ``id`` e ``class``), incluindo vários atributos (o elemento função pode demorar vários argumentos nomeados), mas apenas o primeiro elemento correspondente será devolvido.

Usando o jQuery sintaxe “div # target” é possível especificar vários critérios de pesquisa separadas por uma vírgula:

```
a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))
d = a.elements('span#t1, div.c2')
```

ou equivalente

```
a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))
d = a.elements('span#t1', 'div.c2')
```

Se o valor de um atributo é especificado usando um argumento de nome, pode ser uma string ou uma expressão regular:

```
a = DIV(SPAN('a', _id='test123'), DIV('b', _class='c2'))
d = a.elements('span', _id=re.compile('test\d{3}'))
```

Um especial chamado argumento do DIV (e derivados) helpers é ``find``. Ele pode ser usado para especificar um valor de pesquisa ou uma expressão regular de busca no conteúdo de texto do tag. Por exemplo:

```
>>> a = DIV(SPAN('abcde'), DIV('fghij'))
>>> d = a.elements(find='bcd')
>>> print d[0]
<span>abcde</span>
```

ou

```
>>> a = DIV(SPAN('abcde'), DIV('fghij'))
>>> d = a.elements(find=re.compile('fg\w{3}'))
>>> print d[0]
<div>fghij</div>
```

## 10.5.2 ``Components``

Aqui está um exemplo de listar todos os elementos em uma string html:

```
>>> html = TAG('<a>xxx</a><b>yyy</b>')
>>> for item in html.components:
...     print item
...
<a>xxx</a>
<b>yyy</b>
```

## 10.5.3 `` ``Parent`` e siblings``

``Parent`` retorna o pai do elemento de corrente.

```
>>> a = DIV(SPAN('a'), DIV('b'))
>>> s = a.element('span')
>>> d = s.parent
>>> d['_class']='abc'
>>> print a
<div class="abc"><span>a</span><div>b</div></div>
>>> for e in s.siblings(): print e
<div>b</div>
```

### 10.5.4 Substituir elementos

Elementos que são combinados também podem ser substituídos ou removidos, especificando o argumento ``replace``. Observe que uma lista dos elementos correspondentes originais ainda é devolvido como de costume.

```
>>> a = DIV(SPAN('x'), DIV(SPAN('y'))
>>> b = a.elements('span', replace=P('z'))
>>> print a
<div><p>z</p><div><p>z</p></div>
```

``Replace`` pode ser um que pode ser chamado. Neste caso, será passado o elemento original e é esperado para retornar o elemento de substituição:

```
>>> a = DIV(SPAN('x'), DIV(SPAN('y'))
>>> b = a.elements('span', replace=lambda t: P(t[0]))
>>> print a
<div><p>x</p><div><p>y</p></div>
```

Se ``substituir = None``, os elementos correspondentes serão completamente removidas.

```
>>> a = DIV(SPAN('x'), DIV(SPAN('y'))
>>> b = a.elements('span', replace=None)
>>> print a
<div></div>
```

### 10.5.5 ``flatten``

O método `flatten` recursivamente serializa o conteúdo dos filhos de um determinado elemento em texto normal (sem etiquetas):

```
>>> a = DIV(SPAN('this', DIV('is', B('a'))), SPAN('test'))
>>> print a.flatten()
thisisatest
```

Nívelar pode ser passado um argumento opcional, ``render``, isto é, uma função que processa / achata o conteúdo utilizando um protocolo diferente. Aqui está um exemplo para serializar algumas tags em sintaxe Markmin wiki:

```
>>> a = DIV(H1('title'), P('example of a ', A('link', _href='#test'))
>>> from gluon.html import markmin_serializer
>>> print a.flatten(render=markmin_serializer)
# titles
example of *a link *
```

No momento da escrita nós fornecemos ``markmin\_serializer`` e ``markdown\_serializer``.

### 10.5.6 Análise

O objeto TAG é também um XML / HTML parser. Pode ler o texto e converter em uma estrutura de árvore de helpers. Isto permite a manipulação usando a API acima:

```
>>> html = '<h1>Title</h1><p>this is a <span>test</span></p>'
>>> parsed_html = TAG(html)
>>> parsed_html.element('span')[0]='TEST'
>>> print parsed_html
<h1>Title</h1><p>this is a <span>TEST</span></p>
```

## 10.6 Layout da página

Visualizações pode estender e incluir outros pontos de vista em uma estrutura de árvore.

Por exemplo, podemos pensar em uma visão “index.html” que se estende “layout.html” e inclui “body.html”. Ao mesmo tempo, “layout.html” pode incluir “header.html” e “footer.html”.

A raiz da árvore é o que chamamos de exibição de layout. Assim como qualquer outro arquivo de modelo HTML, você pode editá-lo usando a interface administrativa py4web. O nome do arquivo “layout.html” é apenas uma convenção.

Aqui está uma página minimalista que se estende a visão “layout.html” e inclui o ponto de vista “page.html”:

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

O arquivo de layout estendido deve conter um “[[incluir]]” directiva, algo como:

```
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    [[include]]
  </body>
</html>
```

Quando o ponto de vista é chamado, o (layout) vista alargada é carregado, e o ponto de vista chamando substitui a “[[incluir]]” directiva dentro da disposição. O processamento continua de forma recursiva até que todo “extend” e directivas include” tenham sido processados. O modelo resultante é então traduzido em código Python. Note, quando um aplicativo é bytecode compilado, é este código Python que é compilado, não a visão original próprios arquivos. Assim, a versão bytecode compilado de um determinado ponto de vista é um único arquivo .pyc que inclui o código Python não apenas para o arquivo de exibição original, mas para toda a sua árvore de pontos de vista estendidas e incluídos.

“Extend”, “include”, “block” e “super” são directivas especiais do template, e não comandos Python.

Qualquer conteúdo ou código que precede a “[[estender ...]]” directiva será inserido (e, portanto, executado) antes do início do conteúdo / código da vista estendida. Embora este não é normalmente usado para inserir conteúdo HTML real antes de o conteúdo da exibição estendida, ele pode ser útil como um meio para definir variáveis ou funções que você deseja disponibilizar para a exibição estendida. Por exemplo, considere uma visão “index.html”:



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

e um trecho de “layout.html”:

```
[[if sidebar_enabled:]]
    <div id="sidebar">
        Sidebar Content
    </div>
[[pass]]
```

Porque o `` atribuição `sidebar_enabled``` em “index.html” vem antes do `` `extend```, essa linha é inserido antes do início do “layout.html”, fazendo com que `` qualquer lugar `sidebar_enabled``` disponível dentro do “layout.html” código (uma versão um pouco mais sofisticada deste é usado no bem-vindo `*** app`).

Também é importante ressaltar que as variáveis retornadas pela função de controlador estão disponíveis não só na vista principal da função, mas em todos os seus pontos de vista estendidas e incluídos também.

O argumento de um `` `` `extend``` ou `include``` (isto é, o nome vista alargada ou incluído) possa ser uma variável Python (embora não uma expressão Python). No entanto, este impõe uma limitação - vistas que usar variáveis no `` `extend``` ou `` declarações `include``` não pode ser bytecode compilado. Como mencionado acima, vista compilado-bytecode incluem toda a árvore de pontos de vista estendidas e incluídos, de modo que o específica estendida e vistas incluídos deve ser conhecido em tempo de compilação, que não é possível se os nomes de exibição são variáveis (cujos valores não são determinados até run Tempo). Porque vistas bytecode compilação pode fornecer um impulso de velocidade significativa, utilizando variáveis em `` `extend``` e `` `include``` geralmente deve ser evitada, se possível.

Em alguns casos, uma alternativa para usar uma variável em um `` `include``` é simplesmente para colocar regulares `` `[[incluem ...]]``` directivas dentro de um `` `se ... bloco else```.

```
[[if some_condition:]]
[[include 'this_view.html']]
[[else:]]
[[include 'that_view.html']]
[[pass]]
```

O código acima não apresenta qualquer problema para a compilação bytecode porque há variáveis estão envolvidas. Note, no entanto, que o bytecode compilado vista realmente irá incluir o código Python para ambos “this\_view.html” e “that\_view.html”, embora apenas o código para um desses pontos de vista serão executadas, dependendo do valor de `` `some_condition```.

Tenha em mente, isso só funciona para `` `include``` - você não pode colocar `` `[[estender ...]]``` directivas dentro `` `se ... blocos else```.

Layouts são usados para página encapsular comunalidade (cabeçalhos, rodapés, menus), e embora eles não são obrigatórios, eles vão fazer a sua aplicação mais fácil de escrever e manter. Em particular, sugerimos escrever layouts que aproveitam as seguintes variáveis que podem ser definidas no controlador. Usando estas variáveis bem conhecidas irá ajudar a tornar seus layouts intercambiáveis:

```
response.title
response.subtitle
response.meta.author
response.meta.keywords
response.meta.description
response.flash
response.menu
response.files
```

Exceto para `` menu`` e `` files``, estas são todas as cordas e seu significado deve ser óbvia.

`` Menu response.menu`` está uma lista de 3-tuplas ou 4-tuplas. Os três elementos são: o nome do link, um booleano representando se o link está ativo (é o elo atual), e o URL da página vinculada. Por exemplo:

```
response.menu = [('Google', False, 'http://www.google.com', []),
                  ('Index', True, URL('index'), [])]
```

O quarto elemento tupla é um sub-menu de opcionais.

`` Response.files`` é uma lista de arquivos CSS e JS que são necessários pelo sua página.

Também recomendamos que você usa:

```
[[include 'py4web_ajax.html']]
```

na cabeça HTML, uma vez que irá incluir as bibliotecas jQuery e definir algumas funções JavaScript compatível com versões anteriores para efeitos especiais e Ajax. “Py4web\_ajax.html” inclui os `` tag response.meta`` na vista, base jQuery, o datepicker calendário, e todos CSS necessário e JS `` response.-files``.

### 10.6.1 Layout de página padrão

O “views / layout.html” que acompanha o aplicativo andaimes py4web **\*\* boas-vindas \*\*** (despojado de algumas partes opcionais) é bastante complexa, mas tem a seguinte estrutura:

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8" />
  <title>[response.title or request.application]</title>
  ...
  <script src="[URL('static', 'js/modernizr.custom.js')]"></script>

  [[
    response.files.append(URL('static', 'css/py4web.css'))
    response.files.append(URL('static', 'css/bootstrap.min.css'))
    response.files.append(URL('static', 'css/bootstrap-responsive.min.css'))
    response.files.append(URL('static', 'css/py4web_bootstrap.css'))
  ]]

  [[include 'py4web_ajax.html']]

  [[
    # using sidebars need to know what sidebar you want to use
    left_sidebar_enabled = globals().get('left_sidebar_enabled', False)
    right_sidebar_enabled = globals().get('right_sidebar_enabled', False)
    middle_columns = {0:'span12', 1:'span9', 2:'span6'}[
      (left_sidebar_enabled and 1 or 0)+(right_sidebar_enabled and 1 or 0)]
  ]]

  [[block head]][[end]]
</head>

<body>
  <!-- Navbar ===== -->
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="flash">[response.flash or '']</div>
    <div class="navbar-inner">
      <div class="container">
```

```

    [[=response.logo or '']]
    <ul id="navbar" class="nav pull-right">
        [[='auth' in globals() and auth.navbar(mode="dropdown") or '']]
    </ul>
    <div class="nav-collapse">
        [[if response.menu:]]
        [[=MENU(response.menu)]]
        [[pass]]
    </div><!--/.nav-collapse -->
</div>
</div>
</div><!--/top navbar -->

<div class="container">
    <!-- Masthead ===== -->
    <header class="mastheader row" id="header">
        <div class="span12">
            <div class="page-header">
                <h1>
                    [[=response.title or request.application]]
                    <small>[[=response.subtitle or '']]</small>
                </h1>
            </div>
        </div>
    </header>

    <section id="main" class="main row">
        [[if left_sidebar_enabled:]]
        <div class="span3 left-sidebar">
            [[block left_sidebar]]
            <h3>Left Sidebar</h3>
            <p></p>
            [[end]]
        </div>
        [[pass]]

        <div class="[ [=middle_columns]] ">
            [[block center]]
            [[include]]
            [[end]]
        </div>

        [[if right_sidebar_enabled:]]
        <div class="span3">
            [[block right_sidebar]]
            <h3>Right Sidebar</h3>
            <p></p>
            [[end]]
        </div>
        [[pass]]
    </section><!--/main-->

    <!-- Footer ===== -->
    <div class="row">
        <footer class="footer span12" id="footer">
            <div class="footer-content">
                [[block footer]] <!-- this is default footer -->
                ...
            </div>
        </footer>
    </div>

```

```

        [[end]]
    </div>
</footer>
</div>

</div> <!-- /container -->

<!-- The javascript =====
      (Placed at the end of the document so the pages load faster) -->
<script src="[[URL('static', 'js/bootstrap.min.js')]]"></script>
<script src="[[URL('static', 'js/py4web_bootstrap.js')]]"></script>
[[if response.google_analytics_id:]]
    <script src="[[URL('static', 'js/analytics.js')]]"></script>
    <script type="text/javascript">
        analytics.initialize({
            'Google Analytics':{trackingId:'[[response.google_analytics_id]]'}
        });</script>
[[pass]]
</body>
</html>

```

Existem algumas características deste layout padrão que tornam muito fácil de usar e personalizar:

- Ele é escrito em HTML5 e usa a biblioteca “Modernizr” para compatibilidade com versões anteriores. O layout real inclui algumas declarações condicionais extras exigidos pelo IE e eles são omitidos por brevidade.
- Ele exibe tanto `` response.title`` e `` response.subtitle`` que pode ser definido em um modelo ou um controlador. Se eles não estão definidos, adota o nome do aplicativo como título.
- Ele inclui o arquivo `` py4web\_ajax.html`` no cabeçalho que gerou todas as declarações de importação da ligação e de script.
- Ele usa uma versão modificada do Twitter Bootstrap para layouts flexíveis que funciona em dispositivos móveis e colunas reorganiza para caber telas pequenas.
- Ele usa “analytics.js” para se conectar ao Google Analytics.
- O `` [[= auth.navbar (...)]l`` exibe uma recepção para o usuário atual e links para as funções de autenticação, como login, logout, registro, alteração de senha, etc. dependendo do contexto. `` Auth.navbar`` é uma fábrica auxiliar e a sua saída podem ser manipulados como qualquer outro auxiliar. É colocado em uma expressão para verificar a existência de auth definição, as avalia a expressão `` no caso de auth é indefinido.
- O `` [[= MENU (response.menu)]l`` exibe a estrutura do menu como `` <ul> ... </ul> ``.
- `` [[Incluir]] `` é substituído pelo conteúdo da vista que se prolonga, quando a página é processada.
- Por padrão, ele usa uma de três colunas condicional (a esquerda e barras laterais direitas pode ser desligado com as vistas que se estendem)
- Ele usa as seguintes classes: page-header, principal, rodapé.
- Ele contém os seguintes blocos: cabeça, left\_sidebar, centro, right\_sidebar, rodapé.

Em vista, você pode ativar e personalizar barras laterais da seguinte forma:

```

[[left_sidebar_enabled=True]]
[[extend 'layout.html']]

This text goes in center

```

```
[[block left_sidebar]]
This text goes in sidebar
[[end]]
```

## 10.6.2 Personalizando o layout padrão

Personalizando o layout padrão sem edição é fácil, porque a aplicação de boas-vindas é baseado no Twitter Bootstrap que está bem documentado e suporta temas. Em py4web quatro arquivos estáticos que são relevantes para o estilo:

- “Css / py4web.css” contém estilos py4web específicos
- “Css / bootstrap.min.css” contém o estilo CSS Twitter Bootstrap
- “Css / py4web\_bootstrap.css”, que substitui alguns estilos Bootstrap para se conformar às necessidades py4web.
- “js / bootstrap.min.js”, que inclui as bibliotecas para efeitos de menu, modais, painéis.

Para alterar as cores e imagens de fundo, tente anexar o seguinte código ao header layout.html:

```
<style>
body { background: url('images/background.png') repeat-x #3A3A3A; }
a { color: #349C01; }
.page-header h1 { color: #349C01; }
.page-header h2 { color: white; font-style: italic; font-size: 14px; }
.statusbar { background: #333333; border-bottom: 5px #349C01 solid; }
.statusbar a { color: white; }
.footer { border-top: 5px #349C01 solid; }
</style>
```

Claro que você também pode substituir completamente o “layout.html” e arquivos “py4web.css” com o seu próprio.

## 10.6.3 Desenvolvimento móvel

Embora o layout.html padrão é projetado para ser compatível com telemóvel, pode-se às vezes é preciso usar diferentes pontos de vista quando uma página é visitada por um dispositivo móvel.

Para tornar a desenvolver para desktop e dispositivos móveis mais fáceis, py4web inclui o “@mobilize” decorador. Este decorador é aplicado a ações que devem ter uma visão normal e uma exibição móvel. Isso é demonstrado aqui:

```
from gluon.contrib.user_agent_parser import mobilize
@mobilize
def index():
    return dict()
```

Observe que o decorador deve ser importada antes de usá-lo em um controlador. Quando a função “index” é chamado a partir de um browser normal (computador de mesa), py4web tornará o dicionário retornado usando a exibição “[controller] /index.html”. No entanto, quando ele é chamado por um dispositivo móvel, o dicionário vai ser processado por “[controller] /index.mobile.html”. Observe que visualizações móveis têm a extensão “mobile.html”.

Alternativamente, você pode aplicar a seguinte lógica para fazer todos os pontos de vista móvel amigável:

```
if request.user_agent().is_mobile:
    response.view.replace('.html', '.mobile.html')
```

A tarefa de criar os <quotechar> \*. Mobile.html <quotechar> vista é deixada para o desenvolvedor, mas sugerimos usando o plugin “jQuery Mobile” que torna a tarefa muito fácil.

## 10.7 Funções em vista

Considere isso “layout.html”:

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

e este ponto de vista que se prolonga

```
[[def mysidebar():]]
my new sidebar!!!
[[return]]
[[extend 'layout.html']]
Hello World!!!
```

Repare que a função é definida antes do ``[[estender ...]]`` declaração - Isto resulta na função que está sendo criado antes do código “layout.html” é executado, assim que a função pode ser chamado em qualquer lugar dentro “layout. html”, mesmo antes do ``[[incluir]]``. *Observe também a função está incluída na vista alargada sem a ``=`` prefixo.*

O código gera o seguinte resultado:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
      my new sidebar!!!
    </div>
  </body>
</html>
```

Observe que a função é definida em HTML (embora ele também poderia conter código Python) para que ``response.write`` é usado para gravar o seu conteúdo (a função não retornar o conteúdo). É por isso que o layout chama a função de visão utilizando ``[[mysidebar ()]]`` em vez de ``[[= mysidebar ()]]``. Funções definidas desta forma pode ter argumentos.

## 10.8 Blocos em vista

O caminho principal para fazer uma vista mais modular é usando ``[[bloco ...]]`` s e este mecanismo é uma alternativa para o mecanismo discutido na secção anterior.

Para entender como isso funciona, considere aplicativos baseado no bem-vindo andaimes aplicativo, que tem um layout.html vista. Este ponto de vista é estendida pela vista ``padrão / index.html`` via ``[[es-

tender “layout.html”]] ``. O conteúdo do layout.html predefinir certos blocos com determinado conteúdo padrão, e estes são, portanto, incluídos em default / index.html.

Você pode substituir esses blocos de conteúdo padrão, colocando o seu novo conteúdo dentro do mesmo nome do bloco. A localização do bloco no layout.html não é alterado, mas o conteúdo é.

Aqui está uma versão Simplificado. Imagine isto é “layout.html”:

```
<html>
  <body>
    [[include]]
    <div class="sidebar">
      [[block mysidebar]]
      my default sidebar (this content to be replaced)
    [[end]]
    </div>
  </body>
</html>
```

e isto é um simples que se estende vista `` padrão / index.html``:

```
[[extend 'layout.html']]
Hello World!!!
[[block mysidebar]]
my new sidebar!!!
[[end]]
```

Ele gera a saída seguinte, quando o teor é fornecido pelo bloco sobre-montada na vista estendendo-se, ainda a DIV envolvente e classe vem de layout.html. Isso permite que a consistência entre os pontos de vista:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
      my new sidebar!!!
    </div>
  </body>
</html>
```

O verdadeiro layout.html define um número de blocos úteis, e você pode facilmente adicionar mais para coincidir com o layout seu desejo.

Você pode ter muitos blocos, e se um bloco está presente na exibição estendida, mas não na visão estendendo, o conteúdo da visão ampliada é usado. Além disso, observe que, ao contrário com as funções, não é necessário definir blocos antes do `` [[estender ...]] `` - mesmo se definido após o `` extend``, eles podem ser usados para fazer substituições em qualquer lugar a vista estendida.

Dentro de um bloco, você pode usar a expressão `` [[Super]] `` para incluir o conteúdo do pai. Por exemplo, se substituir o acima estendendo vista com:

```
[[extend 'layout.html']]
Hello World!!!
[[block mysidebar]]
[[super]]
my new sidebar!!!
[[end]]
```

nós temos:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
      my default sidebar
      my new sidebar!
    </div>
  </body>
</html>
```



---

## Internacionalização

---

### 11.1 Pluralizar

Pluralizar é uma biblioteca Python para a Internacionalização (i18n) e Pluralização (p10n).

A biblioteca assume uma pasta (por exaple “traduções”) que contém arquivos como:

```
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)
```

Adicionar um novo idioma suportado (por exemplo alemão “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

---

### TRABALHO EM PROGRESSO

Só sei que ``py4web.utils.form.Form`` é um praticamente equivalente a ``SQLFORM`` do web2py.

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,
      before_validate=None):
```

Onde:

- ``Table``: uma mesa DAL ou uma lista de campos (equivalente a SQLFORM.factory idade)
- ``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``: uma função que processa a forma usando ajudantes (FormStyleDefault)
- ``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
- ``Before\_validate``: um validador opcional.

## 12.1 Exemplo

Aqui está um exemplo simples de um formulário personalizado não utilizar acesso de banco de dados. Declaramos um ponto final `` / form\_example``, que será utilizado tanto para o GET e para o POST da

forma:

```
from py4web import Session, redirect, URL
from py4web.utils.dbstore import DBStore
from py4web.utils.form import Form, FormStyleBulma

db = DAL('sqlite:memory')
session = Session(storage=DBStore(db))

@action('form_example', method=['GET', 'POST'])
@action.uses('form_example.html', session)
def form_example():
    form = Form([
        Field('product_name'),
        Field('product_quantity', 'integer'),
        formstyle=FormStyleBulma)
    if form.accepted:
        # Do something with form.vars['product_name'] and
        form.vars['product_quantity']
        redirect(URL('index'))
    return dict(form=form)
```

A forma pode ser exibida no modelo simplesmente usando ``[[= formar]]``.

## 12.2 Validação de formulário

A validação da entrada de formulário pode ser feito de duas maneiras. Pode-se definir atributos ``requires`` de `Field`, ou pode-se definir explicitamente uma função de validação. Para fazer o último, passamos para ``validate`` uma função que toma a forma e retorna um dicionário, os nomes de campo mapeamento para erros. Se o dicionário não é vazio, os erros serão exibidos para o usuário, e nenhum banco de dados I / O ocorrerá.

Aqui está um exemplo:

```
from py4web import Field
from py4web.utils.form import Form, FormStyleBulma
from pydal.validators import IS_INT_IN_RANGE

def check_nonnegative_quantity(form):
    if not form.errors and form.vars['product_quantity'] % 2:
        form.errors['product_quantity'] = T('The product quantity must be even')

@action('form_example', method=['GET', 'POST'])
@action.uses('form_example.html', session)
def form_example():
    form = Form([
        Field('product_name'),
        Field('product_quantity', 'integer', requires=IS_INT_IN_RANGE(0,100)),
        validation=check_nonnegative_quantity,
        formstyle=FormStyleBulma)
    if form.accepted:
        # Do something with form.vars['product_name'] and
        form.vars['product_quantity']
        redirect(URL('index'))
    return dict(form=form)
```

---

## Autenticação e controle de acesso

---

**\*\* Atenção:** a API descrito neste capítulo é novo e sujeitas a alterações. Certifique-se de manter seu código atualizado \*\*

py4web vem com um objecto Auth e um sistema de encaixes para a autenticação do utilizador e de controlo de acesso. Ele tem o mesmo nome que o web2py correspondente e tem a mesma finalidade, mas a API e design interno é muito diferente.

Para usá-lo, em primeiro lugar você precisa importá-lo, instanciá-lo, configurá-lo, e habilitá-lo.

```
from py4web.utils.auth import Auth
auth = Auth(session, db)
# (configure here)
auth.enable()
```

A etapa de importação é óbvia. A segunda etapa não executar qualquer operação que não dizendo o objeto Auth qual objeto sessão para usar e qual banco de dados para uso. Auth dados são armazenados em ``sessão [ "user" ]`` e, se um usuário estiver logado, o ID do usuário é armazenado na sessão [ 'user' ] [ 'id' ]. O objecto dB é utilizado para armazenar informação sobre o utilizador persistente numa tabela ``auth\_user`` com os seguintes campos:

- 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).

Se o ``auth\_user`` tabela não existir ele será criado.

A etapa de configuração é opcional e discutida mais tarde.

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 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 / static / componentes / auth.js
- \_Scaffold / static / componentes / auth.html

Os arquivos do componente (js / html) definem um componente Vue `` <auth /> `` que é usado na auth.html arquivo de modelo da seguinte forma:

```
[[extend "layout.html"]]
<div id="vue">
  <div class="columns">
    <div class="column is-half is-offset-one-quarter" style="border : 1px solid
#e1e1e1; border-radius: 10px">
      <auth plugins="local,oauth2google,oauth2facebook"></auth>
    </div>
  </div>
</div>
[[block page_scripts]]
<script src="js/utils.js"></script>
<script src="componentes/auth.js"></script>
<script>utils.app().start();</script>
[[end]]
```

Você pode muito bem usar esse arquivo modificado-un. Estende-se o layout atual e incorpora o `` <auth /> `` componente na página. Em seguida, usa `` utils.app () start (); `` (Magia py4web) para processar o conteúdo de `` <div id = «vue»> ... </ div> `` usando Vue.js. `` componentes / auth.js `` também carrega automaticamente `` componentes / auth.html `` para o espaço reservado componente (mais mágicas py4web). O componente é responsável para render o login / registro / etc formas usando reactivo html e Geting / dados de lançamento com a API de serviço de autenticação.

Se você precisa mudar o estilo do componente que você pode editar “componentes / auth.html” para atender às suas necessidades. É principalmente HTML com algum especial Vue `` v- \* `` tags.

## 13.2 Usando o Auth

Há duas maneiras de usar o objeto Auth em uma ação:

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

```
@action.uses(auth)
def index():
    user = auth.get_user()
    return 'hello {first_name}'.format(**user) if user else 'not logged in'
```

Com ``@ action.uses (auth) `` nós dizemos py4web que esta ação precisa ter informações sobre o usuário, em seguida, tentar analisar a sessão para uma sessão de usuário.

```
@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.

## 13.3 Plugins de Autenticação

Plugins são definidos no “py4web / utils / auth\_plugins” e eles têm uma estrutura hierárquica. Alguns são exclusivos e alguns não são. Por exemplo, padrão, LDAP, PAM, e SAML são exclusivos (o desenvolvedor tem que escolher um). Padrão, o Google, Facebook e Twitter OAuth não são exclusivos (o desenvolvedor pode pegar todos eles e que o usuário começa a escolher usando a interface do usuário).

O `` <auth /> `` componentes irá se adaptar automaticamente para formulários de login de exibição, conforme exigido pelos plugins instalados.

**\*\* Neste momento, não podemos garantir que os seguintes plugins funcionam bem. Eles foram portados de web2py onde eles não funcionam, mas o teste ainda é necessária \*\***

### 13.3.1 PAM

Configurando PAM é o mais fácil:

```
from py4web.utils.auth_plugins.pam_plugin import PamPlugin
auth.register_plugin(PamPlugin())
```

Este, como todos os plugins deve ser importado e registrado. Uma vez registrado o UI (componentes / auth) e as APIs RESTful sabe como lidar com isso. O construtor desta plugins não requer quaisquer argumentos (onde outros plugins fazer).

O `` auth.register\_plugin (...) `` must **\*\* \*\*** vir antes do `` auth.enable () `` , uma vez que não faz sentido para expor APIs antes de plugins desejados são montados.

### 13.3.2 LDAP

```
from py4web.utils.auth_plugins.ldap_plugin import LDAPPlugin
LDAP_SETTING = {
    'mode': 'ad',
    'server': 'my.domain.controller',
    'base_dn': 'ou=Users,dc=domain,dc=com'
}
auth.register_plugin(LDAPPlugin(**LDAP_SETTINGS))
```

### 13.3.3 OAuth2 com Google (testado OK)

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

O ID de cliente e segredo do cliente deve ser fornecido pelo Google.

### 13.3.4 OAuth2 com Facebook (testado OK)

```
from py4web.utils.auth_plugins.oauth2facebook import OAuth2Facebook # UNTESTED
auth.register_plugin(OAuth2Facebook(
    client_id=CLIENT_ID,
    client_secret=CLIENT_SECRET,
    callback_url='auth/plugin/oauth2google/callback'))
```

O ID de cliente e segredo do cliente deve ser fornecido pelo Facebook.

## 13.4 Etiquetas e permissões

O Py4web não tem o conceito de grupos como web2py. A experiência mostrou que, enquanto esse mecanismo é poderoso ele sofre de dois problemas: é um exagero para a maioria dos aplicativos, e não é suficiente flexível para aplicativos muito complexos. Py4web fornece um mecanismo de marcação de uso geral que permite ao desenvolvedor tag qualquer registro de qualquer tabela, verificar a existência de marcas, bem como a verificação de registros contendo uma tag. associação de grupo pode ser considerado um tipo de tag que se aplicam aos usuários. As permissões também pode ser tags. Desenvolvedor é livre para criar sua própria lógica no topo do sistema de marcação.

Para usar o sistema de marcação, você precisa criar um objeto para marcar uma tabela:

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

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')
```

Aqui, o caso de uso é o controle de acesso baseado em grupo, onde o desenvolvedor primeiro verifica se um usuário é um membro do «grupo`manager”», se o usuário não é um administrador (ou ninguém está logado) redirecionamentos py4web ao `` “não autorizada url””. Se o usuário estiver no grupo correto, então manager Olá “py4web exibe:

```
@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}
```

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'
```

**\*\* Importante: \*\*** ``Tags`` são automaticamente hierárquica. Por exemplo, se um usuário tem um grupo tag 'professor /-ensino médio / física', em seguida, todas as seguintes seaches retornará o usuário:

- `` Groups.find ( "professor /-ensino médio / física") ``
- `` Groups.find ( "professor /-colegial") ``
- `` Groups.find ( "professor") ``

Isto significa que as barras têm um significado especial para tags. Slahes no início ou no final de uma tag são opcionais. Todos os outros caracteres são permitidos em pé de igualdade.

Observe que uma tabela pode ter vários associados ``Tags`` objetos. Os grupos nome aqui é completamente arbitrária, mas tem um significado semântico específico. Diferentes ``objectos Tags`` são ortogonais entre si. O limite para o seu uso é a sua criatividade.

Por exemplo, você poderia criar um grupos de mesa:

```
db.define_table('auth_group', Field('name'), Field('description'))
```

e Tags:

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

Em seguida, crie um grupo zipper, dar-lhe uma permissão, e tornar um membro do usuário do grupo:

```
zap_id = db.auth_group.insert(name='zipper', description='can zap database')
permissions.add(zap_id, 'zap database')
groups.add(user.id, 'zipper')
```

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.

py4web vem com um objecto de grelha fornecendo grid simples e os recursos CRUD.

## 14.1 Características principais

- 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
- CRUD completa com Confirmação de exclusão
- 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

## 14.2 Exemplo básico

Neste exemplo simples, vamos fazer uma grid sobre a mesa da empresa.

controllers.py

```
from functools import reduce
from py4web.utils.grid import Grid
from py4web import action
from .common import db, session, auth

@action('companies', method=['POST', 'GET'])
@action('companies/<path:path>', method=['POST', 'GET'])
@action.uses(session, db, auth, 'grid.html')
def companies(path=None):
    grid = Grid(path,
                query=reduce(lambda a, b: (a & b), [db.company.id > 0]),
                orderby=[db.company.name],
                search_queries=[['Search by Name', lambda val:
db.company.name.contains(val)]])
```

```
return dict(grid=grid)
```

grid.html

```
[[extend 'layout.html']]  
[[=grid.render()]]
```

## 14.3 Assinatura

```
class Grid:  
    def __init__(  
        self,  
        path,  
        query,  
        search_form=None,  
        search_queries=None,  
        fields=None,  
        show_id=False,  
        orderby=None,  
        left=None,  
        headings=None,  
        create=True,  
        details=True,  
        editable=True,  
        deletable=True,  
        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,  
    ):
```

- caminho: a rota do pedido
- query: consulta pydal a ser processado
- search\_form: FORM py4web a ser incluído como o formulário de busca. Se search\_form é passado em seguida, o desenvolvedor é responsável por aplicar o filtro para a consulta passada. Isso é diferente de search\_queries.
- search\_queries: lista de listas de consulta para usar para construir o formulário de busca. Ignorado se search\_form é usado. O formato é
- campos: lista de campos a serem exibidos na página de lista, se em branco, tablename recolher de primeira consulta e usar todos os campos dessa tabela
- show\_id: mostrar o campo ID de registro 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
- detalhes: URL para redirecionar para os registros exibindo - Defina como true para gerar automati-

camente a URL - definido como falso para não exibir o botão

- `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
- `editável`: URL para redirecionar para a edição de registros - Defina como true para gerar automaticamente a URL - definido como falso para não exibir o botão
- `deletable`: URL para redirecionar para a exclusão de registros - Defina como true para gerar automaticamente a URL - definido como falso para não exibir o botão
- `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`: objeto `GridClassStyle` usado para os padrões de substituição para denominar sua grid prestados. Permite especificar classes ou estilos para aplicar em certos pontos da grid.

## 14.4 Searching / Filtering

Há duas maneiras de construir um formulário de pesquisa.

- Fornecer uma lista `search_queries`
- Construa a sua própria forma de pesquisa personalizada

Se você fornecer uma lista `search_queries` à grid, ele irá:

- construir um formulário de busca. Se mais de uma consulta de pesquisa na lista, que também irá gerar uma lista suspensa para selecionar qual campo de pesquisa para procurar agains
- 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.

## 14.5 CRUD

A grid oferece recursos CRUD (CRUD) utilizando formulário py4web.

Você pode desligar CRUD apresenta pela configuração `criar` / `details` / `editável` / `elimináveis` durante a instanciação grid.

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.

## 14.6 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()]]
```

Para permitir a personalização de layout do formulário CRUD (como com web2py) você pode usar o seguinte

```
[[form = grid.render() ]]  
[[form.custom["begin"] ]]  
...  
[[form.custom["submit"]]  
[[form.custom["end"]]
```

Ao manusear formulário personalizado layouts que você precisa saber se você está exibindo a grid ou um formulário. Use o seguinte para decidir

```
[[if 'action' in request.url_args and request.url_args['action'] in ['details',  
'edit']:]]  
    # Display the custom form  
    [[form = grid.render() ]]  
    [[form.custom["begin"] ]]  
    ...  
    [[form.custom["submit"]]  
    [[form.custom["end"]]  
[[else:]]  
    [[grid.render() ]]  
[[pass]]
```

## 14.7 Personalizando Estilo

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.

O GridClassStyle padrão - baseado em no.css, principalmente usa estilos para modificar o layout da grid

```
class GridClassStyle:  
  
    """  
    Default grid style  
    Internal element names match default class name, other classes can be added  
    Style use should be minimized since it cannot be overridden by CSS  
    """  
  
    classes = {  
        "grid-wrapper": "grid-wrapper",  
        "grid-header": "grid-header",
```

```

"grid-new-button": "grid-new-button info",
"grid-search": "grid-search",
"grid-table-wrapper": "grid-table-wrapper",
"grid-table": "grid-table",
"grid-sorter-icon-up": "grid-sort-icon-up fas fa-sort-up",
"grid-sorter-icon-down": "grid-sort-icon-down fas fa-sort-down",
"grid-th-action-button": "grid-col-action-button",
"grid-td-action-button": "grid-col-action-button",
"grid-tr": "",
"grid-th": "",
"grid-td": "",
"grid-details-button": "grid-details-button info",
"grid-edit-button": "grid-edit-button info",
"grid-delete-button": "grid-delete-button info",
"grid-footer": "grid-footer",
"grid-info": "grid-info",
"grid-pagination": "grid-pagination",
"grid-pagination-button": "grid-pagination-button info",
"grid-pagination-button-current": "grid-pagination-button-current default",
"grid-cell-type-string": "grid-cell-type-string",
"grid-cell-type-text": "grid-cell-type-text",
"grid-cell-type-boolean": "grid-cell-type-boolean",
"grid-cell-type-float": "grid-cell-type-float",
"grid-cell-type-int": "grid-cell-type-int",
"grid-cell-type-date": "grid-cell-type-date",
"grid-cell-type-time": "grid-cell-type-time",
"grid-cell-type-datetime": "grid-cell-type-datetime",
"grid-cell-type-upload": "grid-cell-type-upload",
"grid-cell-type-list": "grid-cell-type-list",
# specific for custom form
"search_form": "search-form",
"search_form_table": "search-form-table",
"search_form_tr": "search-form-tr",
"search_form_td": "search-form-td",
}

styles = {
    "grid-wrapper": "",
    "grid-header": "display: table; width: 100%",
    "grid-new-button": "display: table-cell;",
    "grid-search": "display: table-cell; float:right",
    "grid-table-wrapper": "overflow-x: auto; width:100%",
    "grid-table": "",
    "grid-sorter-icon-up": "",
    "grid-sorter-icon-down": "",
    "grid-th-action-button": "",
    "grid-td-action-button": "",
    "grid-tr": "",
    "grid-th": "white-space: nowrap; vertical-align: middle",
    "grid-td": "white-space: nowrap; vertical-align: middle",
    "grid-details-button": "margin-bottom: 0",
    "grid-edit-button": "margin-bottom: 0",
    "grid-delete-button": "margin-bottom: 0",
    "grid-footer": "display: table; width:100%",
    "grid-info": "display: table-cell;",
    "grid-pagination": "display: table-cell; text-align:right",
    "grid-pagination-button": "min-width: 20px",
    "grid-pagination-button-current": "min-width: 20px; pointer-events:none;
opacity: 0.7",
    "grid-cell-type-string": "white-space: nowrap; vertical-align: middle;

```

```
text-align: left; text-overflow: ellipsis; max-width: 200px",
    "grid-cell-type-text": "vertical-align: middle; text-align: left;
text-overflow: ellipsis; max-width: 200px",
    "grid-cell-type-boolean": "white-space: nowrap; vertical-align: middle;
text-align: center",
    "grid-cell-type-float": "white-space: nowrap; vertical-align: middle;
text-align: right",
    "grid-cell-type-int": "white-space: nowrap; vertical-align: middle;
text-align: right",
    "grid-cell-type-date": "white-space: nowrap; vertical-align: middle;
text-align: right",
    "grid-cell-type-time": "white-space: nowrap; vertical-align: middle;
text-align: right",
    "grid-cell-type-datetime": "white-space: nowrap; vertical-align: middle;
text-align: right",
    "grid-cell-type-upload": "white-space: nowrap; vertical-align: middle;
text-align: center",
    "grid-cell-type-list": "white-space: nowrap; vertical-align: middle;
text-align: left",
    # specific for custom form
    "search_form": "",
    "search_form_table": "",
    "search_form_tr": "",
    "search_form_td": "",
}

@classmethod
def get(cls, element):
    """returns a dict with _class and _style for the element name"""
    return {
        "_class": cls.classes.get(element),
        "_style": cls.styles.get(element),
    }
```

### GridClassStyleBulma - implementação bulma

```
class GridClassStyleBulma(GridClassStyle):
    classes = {
        "grid-wrapper": "grid-wrapper field",
        "grid-header": "grid-header pb-2",
        "grid-new-button": "grid-new-button button",
        "grid-search": "grid-search is-pulled-right pb-2",
        "grid-table-wrapper": "grid-table-wrapper table_wrapper",
        "grid-table": "grid-table table is-bordered is-striped is-hoverable
is-fullwidth",
        "grid-sorter-icon-up": "grid-sort-icon-up fas fa-sort-up is-pulled-right",
        "grid-sorter-icon-down": "grid-sort-icon-down fas fa-sort-down
is-pulled-right",
        "grid-th-action-button": "grid-col-action-button is-narrow",
        "grid-td-action-button": "grid-col-action-button is-narrow",
        "grid-tr": "",
        "grid-th": "",
        "grid-td": "",
        "grid-details-button": "grid-details-button button is-small",
        "grid-edit-button": "grid-edit-button button is-small",
        "grid-delete-button": "grid-delete-button button is-small",
        "grid-footer": "grid-footer",
        "grid-info": "grid-info is-pulled-left",
        "grid-pagination": "grid-pagination is-pulled-right",
```



```

        "grid-pagination-button": "grid-pagination-button button is-small",
        "grid-pagination-button-current": "grid-pagination-button-current button
is-primary is-small",
        "grid-cell-type-string": "grid-cell-type-string",
        "grid-cell-type-text": "grid-cell-type-text",
        "grid-cell-type-boolean": "grid-cell-type-boolean has-text-centered",
        "grid-cell-type-float": "grid-cell-type-float",
        "grid-cell-type-int": "grid-cell-type-int",
        "grid-cell-type-date": "grid-cell-type-date",
        "grid-cell-type-time": "grid-cell-type-time",
        "grid-cell-type-datetime": "grid-cell-type-datetime",
        "grid-cell-type-upload": "grid-cell-type-upload",
        "grid-cell-type-list": "grid-cell-type-list",
        # specific for custom form
        "search_form": "search-form is-pulled-right pb-2",
        "search_form_table": "search-form-table",
        "search_form_tr": "search-form-tr",
        "search_form_td": "search-form-td pr-1",
    }

    styles = {
        "grid-wrapper": "",
        "grid-header": "",
        "grid-new-button": "",
        "grid-search": "",
        "grid-table-wrapper": "",
        "grid-table": "",
        "grid-sorter-icon-up": "",
        "grid-sorter-icon-down": "",
        "grid-th-action-button": "",
        "grid-td-action-button": "",
        "grid-tr": "",
        "grid-th": "text-align: center; text-transform: uppercase;",
        "grid-td": "",
        "grid-details-button": "",
        "grid-edit-button": "",
        "grid-delete-button": "",
        "grid-footer": "padding-top: .5em;",
        "grid-info": "",
        "grid-pagination": "",
        "grid-pagination-button": "margin-left: .25em;",
        "grid-pagination-button-current": "margin-left: .25em;",
        "grid-cell-type-string": "",
        "grid-cell-type-text": "",
        "grid-cell-type-boolean": "",
        "grid-cell-type-float": "",
        "grid-cell-type-int": "",
        "grid-cell-type-date": "",
        "grid-cell-type-time": "",
        "grid-cell-type-datetime": "",
        "grid-cell-type-upload": "",
        "grid-cell-type-list": "",
        # specific for custom form
        "search_form": "",
        "search_form_table": "",
        "search_form_tr": "",
        "search_form_td": "",
    }

```

Você pode construir seu próprio `class_style` para ser usado com o quadro css de sua escolha.

## 14.8 Ação personalizada Botões

Tal como acontece com web2py, você pode adicionar botões adicionais para cada linha em sua grid. Você faz isso fornecendo `pre_action_buttons` ou `post_action_buttons` à Rede `** inicialização **` método.

- `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

Você pode construir sua própria classe de ação do botão para passar para pré / botões de ação pós baseados no template abaixo (isso não é fornecido com py4web)

## 14.9 Botão Classe Ação Amostra

```
def __init__(self,
              url,
              text,
              icon="fa-calendar",
              additional_classes=None,
              message=None,
              append_id=False):
```

- `url`: a página para navegar até quando o botão é clicado
- `texto`: texto para exibição no botão
- `ícone`: o ícone font-incrível para exibição antes do texto
- `additional_classes`: uma lista separada por espaços de aulas para incluir no elemento botão
- `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

## 14.10 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` na definição de campo PyDAL - aqui está um exemplo de um campo de chave estrangeira

```
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

esquerda juntar-se e especificar campos da tabela juntou - especificado no parâmetro esquerdo da grid instânciação

```
db.company.on(db.employee.company == db.company.id)
```

Você pode especificar um PyDAL padrão LEFT JOIN, incluindo uma lista de junta a considerar.

Agora o campo nome da empresa pode ser incluído em sua lista de campos pode ser clicado e ordenados.

Agora você também pode especificar uma consulta como:

```
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.



---

## De web2py para py4web

---

Este capítulo é dedicado a ajudar os usuários para portar aplicativos web2py antigos para py4web.

---

**Note** Trabalho em progresso

---

- : Ref: *genindex*
- : Ref: *modindex*
- : Ref: *search*





