



{{ Desenvolvimento web com Python e web2py }}



Módulo 1 – Básico

Objetivo: introduzir os alunos ao desenvolvimento web com Python, utilizando o web2py como ferramenta, e apresentar todas as vantagens da utilização de um framework. Para acompanhar o treinamento, os pré-requisitos necessários são conhecimentos básicos de HTML, CSS e lógica de programação (em qualquer linguagem).

>>> for página in páginas: print página

Introdução	4
O que é web2py?	4
Princípios	6
Conceitos básicos da linguagem Python	7
Executando o Python	7
Modo interativo	7
Olá Mundo em Python	8
Cálculos	8
Objetos	9
Tipos dinâmicos	10
Interpolação de variáveis em strings	11
Estruturas de dados	11
<i>Listas</i>	11
<i>Sintaxe</i>	11
<i>Operações com listas</i>	12
<i>Tuplas</i>	12
<i>Dicionários</i>	13
<i>Sobre endentação</i>	15
<i>Iteração de elementos</i>	15
<i>Expressões condicionais</i>	17
<i>lambda</i>	18
<i>Executando código pré-determinado</i>	18
<i>Comentários e docstrings</i>	18
<i>Módulos</i>	19
<i>List Comprehensions</i>	20
WSGI	21
O Padrão MVC	22
Preparando o ambiente de desenvolvimento	23
Criando sua primeira aplicação	25
Camadas, módulos e pacotes do web2py	28
A partir do zero	28
Modelos (models)	29
Controladores (controllers)	32
Visões (views)	33
<i>Criando uma visão</i>	34
<i>Passagem de argumentos para a visão</i>	35
ciclo de vida da aplicação	36
A camada de acesso a dados - DAL	37
<i>Representação de registros</i>	39
<i>Interagindo com a DAL</i>	40
<i>Insert, Select, Update, Delete</i>	41
<i>Query</i>	42
<i>Atalhos</i>	43
<i>Ordenação</i>	44
<i>Query múltipla</i>	44
<i>Contagem de registros</i>	44
<i>Alteração de registros</i>	44
<i>Tipos de dados</i>	45
<i>Atributos</i>	45
Migrações	46
Validadores	46

{{ Desenvolvimento web com Python e web2py }}	
<i>Os validadores de formulário</i>	46
<i>Os validadores de banco de dados</i>	48
Auto complete widget	49
Mão na Massa - Visão Geral.....	53
<i>O Projeto</i>	53
<i>Loja de Carro</i>	53
API.....	63
<i>Objetos globais:</i>	63
<i>Navegação:</i>	63
<i>internacionalização (i18n):</i>	63
<i>Helpers (bibliotecas auxiliares):</i>	63
<i>Validadores:</i>	63
<i>Banco de dados:</i>	64
Autenticação e Controle de Acesso.....	69
<i>Autenticação</i>	70
<i>Restringindo acesso</i>	72
<i>Restrições no registro de usuários</i>	72
<i>Integração com autenticação externa: Facebook, Google, Twitter etc.</i>	72
<i>Autorização</i>	73
<i>Formulários</i>	78
<i>Validação de formulários</i>	80
<i>CRUD</i>	84
jQuery	86
<i>Ajax</i>	87
web2py na web	88
<i>Onde encontrar ajuda:</i>	88
<i>Onde contribuir:</i>	88
<i>Recursos:</i>	88
Instrutores:	89
<i>Bruno Cezar Rocha</i>	89
<i>Álvaro Justen</i>	89

Introdução

O que é web2py?

web2py é um framework para desenvolvimento Web escrito em Python, software livre e gratuito, que tem como um de seus principais objetivos proporcionar agilidade no desenvolvimento de aplicações web seguras, baseadas em bancos de dados.

O framework segue o modelo MVC (Model-View-Controller), que permite melhor organização do código. Ele também é autocontido, ou seja, tudo o que você precisa para desenvolver alguma aplicação está nele, basta baixar e descompactar para começar - nada de configurações!

Com o foco em permitir que o desenvolvedor pense apenas na aplicação que está desenvolvendo, o web2py possui integração com mais de 10 sistemas de banco de dados e vários subsistemas, como: criação automática de formulários com validação automática; autenticação e autorização; gerador de códigos AJAX para melhor interação do usuário com a aplicação; upload seguro de arquivos; sistema de plugins; integração com vários padrões web (XML, RSS etc.), dentre outros.

O web2py leva em consideração todas as questões referentes à segurança da aplicação web, e isso significa que o framework se preocupa em tratar as vulnerabilidades aplicando práticas bem estabelecidas, como, por exemplo, validando formulários com prevenção de injeção de código malicioso; efetuando o correto *escape* da saída HTML para prevenir ameaças como o *cross-site scripting* e renomeando os arquivos de upload utilizando hash seguro. O web2py toma conta, automaticamente, de questões de segurança.

- Segurança do web2py www.pythonsecurity.org/wiki/web2py/

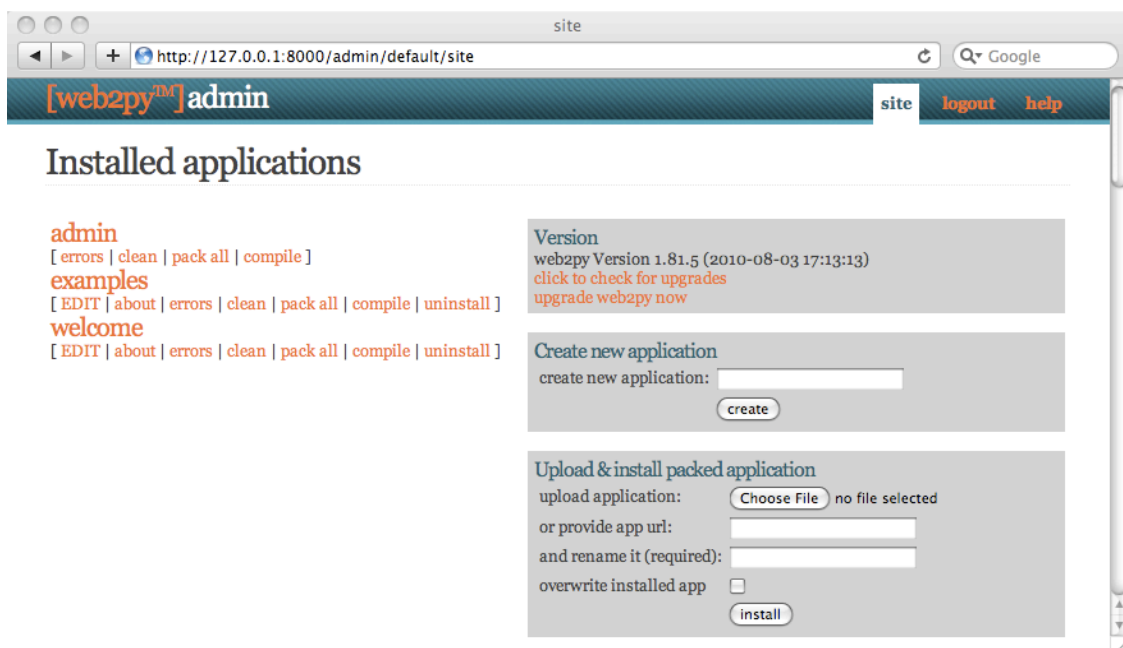
O web2py possui uma camada de abstração de banco de dados (**DAL**) que gera dinamicamente todo o código SQL. A classe **DAL** pode se conectar e lidar de forma transparente com os bancos de dados:

- SQLite
- MySQL
- PostgreSQL
- MSSQL
- FireBird
- Oracle
- IBM DB2
- Informix
- Ingres
- Google BigTable on Google App engine **GAE**

A partir da definição de um ou mais bancos de dados para uma aplicação, o web2py gera uma interface administrativa básica para acesso aos dados, não necessitando o uso de ferramentas para executar tarefas *de administração*. Em uma única aplicação, o web2py pode se conectar com vários bancos de dados ao mesmo tempo, podendo facilmente integrar dados entre MySQL e Oracle, por exemplo.

A aplicação de administração, que acompanha a instalação padrão do framework, provê uma interface administrativa que oferece todos os recursos para o desenvolvimento web:

- Criação de novas aplicações
- Edição de códigos Python, HTML, JavaScript
- Upload de arquivos estáticos como imagens e pacotes compactados
- Gerenciamento de arquivo de recurso de linguagem
- Sistema de tickets para acompanhamento do log de exceções geradas na aplicação
- Atualização automática do framework
- Interface para o sistema de controle de versões (Mercurial)



Podendo ser executado com **CPython** (a implementação padrão da linguagem Python implementada em C) ou com **Jython** (a implementação do Python escrita em Java), o web2py funciona com quase todos os seus recursos na plataforma *gratuita* de hospedagem de aplicações do Google, o GAE (Google App Engine).

O web2py é Open Source sob a licença GPL2, mas as aplicações desenvolvidas com o web2py não estão sujeitas a este tipo de licenciamento, ou seja, você pode desenvolver suas aplicações web2py e licenciar da maneira que escolher, seja open source ou software comercial. O web2py oferece a possibilidade de compilar as aplicações em *byte-code* para distribuição com código fechado.

Outra característica do web2py é o compromisso com a compatibilidade em versões futuras - compromisso que tem sido mantido desde a primeira versão. Isso significa que as aplicações desenvolvidas em uma versão específica do framework funcionarão em versões mais recentes, sem que seja necessária nenhuma alteração de código.

Princípios

A linguagem Python tipicamente segue os seguintes princípios

- Não se repita (DRY)
- Deve existir apenas uma maneira óbvia para fazer algo
- Explícito é melhor do que implícito

O web2py procura seguir rigorosamente os dois primeiros princípios, para estimular o desenvolvedor a utilizar boas práticas de engenharia de software, seguindo um bom padrão de projeto (**MVC**) e incentivando o reaproveitamento de código. O framework guia o desenvolvedor, facilitando as tarefas mais comuns em desenvolvimento para web.

Diferente de outros frameworks, o web2py aplica parcialmente o terceiro princípio. Com a intenção de prover o desenvolvimento ágil, o framework efetua configurações, importa módulos e instancia objetos globais como *session*, *request*, *cache*, *Translation*. Tudo isso é feito automaticamente, com a intenção de evitar, ao máximo, que o desenvolvedor tenha que importar os mesmos módulos e instanciar os mesmos objetos no início de cada *model* ou *controller*.

Efetuando automaticamente estas operações, o web2py evita problemas e mantém o princípio de *não se repetir* e de que *deve existir apenas uma maneira para se fazer algo*. Porém, se o desenvolvedor desejar, este comportamento pode ser alterado, possibilitando que em qualquer cenário seja possível importar módulos (como ocorre em qualquer outra aplicação ou framework Python).

Este treinamento objetiva introduzir os alunos ao desenvolvimento web com Python, utilizando o web2py como ferramenta, e pretender apresentar todas as vantagens da utilização de um framework. Para acompanhar o treinamento, os pré-requisitos necessários são conhecimentos básicos de HTML, CSS e lógica de programação (em qualquer linguagem).

Conceitos básicos da linguagem Python



é uma linguagem de programação dinâmica de altíssimo nível, orientada a objetos, interpretada, interativa e de tipagem dinâmica e forte. É utilizada em larga escala por empresas como Google, Yahoo, Dreamworks e Industrial Light & Magic. No Brasil, é utilizada pela Locaweb, Globo.com, SERPRO, Interlegis (órgão vinculado ao Senado Federal), entre outros. Diversos softwares como GIMP, Inkscape e Blender3D utilizam a linguagem Python para extensões e criação de plugins.

Abordaremos apenas os conceitos básicos da linguagem, para apresentá-la àqueles que estão iniciando no mundo da programação, ou àqueles que já programam, mas em outras linguagens. Para uma abordagem mais profunda de todos os aspectos da linguagem, é sugerida a leitura do material encontrado no site oficial da linguagem:

<http://www.python.org.br>

Executando o Python

Esta apostila não abordará a instalação do interpretador Python. A maioria dos sistemas operacionais baseados em Unix, como Ubuntu, MacOS e outras distribuições Linux, já costumam vir com Python instalado. O Python também pode ser instalado/executado em qualquer versão do Windows; para isso, basta seguir as dicas de instalação encontradas em <http://www.python.org.br>

Considerando que seu sistema operacional já esteja pronto para executar o interpretador Python em modo interativo, inicie uma janela de terminal Linux/Unix/MacOS, ou um prompt de comando no Windows, e digite:

```
1. python
```

Modo interativo

Ao iniciar o interpretador **Python**, será carregado um console interativo. Qualquer comando digitado neste console será interpretado pelo **Python**.

```
1. Python 2.6.5 (r265:79063), Apr 16 2010, 13:09:56)
2. [GCC 4.4.3 on linux2
3. Type "Help", "copyright", "credits" or "licence" for more information.
4. >>>
```

Olá Mundo em Python

O comando **print** imprime qualquer objeto que seja do tipo **string** ou que possa ser serializado como string. Isso vale para textos, números, objetos, estruturas de dados e muito mais.

1. Python 2.6.5 (r265:79063), Apr 16 2010, 13:09:56)
2. [GCC 4.4.3 on linux2
3. Type "Help", "copyright", "credits" or "licence" for more information.
4. >>> **print** "Olá Mundo"
5. Olá Mundo

No console interativo não é obrigatório utilizar a função **print** para imprimir alguma coisa, pois neste modo o Python sempre responde com o retorno padrão do objeto. Portanto, omitindo o comando print...

1. >>> 'Olá Mundo'
2. Olá Mundo

... teremos o mesmo resultado.

Mas fique atento, sempre que escrevemos programas em arquivos .py, é necessário utilizar a função **print**.

Cálculos

O Console Python responde a operadores matemáticos da mesma forma que uma calculadora.

1. >>> (5+2)*9
2. 63
3. >>> 50+50
4. 100
5. >>> 7/2
6. 3 # Ops tem algo errado, 7/2 não é 3.5? Sim mas neste caso Python arredonda para baixo
7. >>> 7.0/2
8. 3.5 # passando um número do tipo **float** o retorno também será **float**
9. >>> "Brasil "*3 # Python também multiplica **strings**
10. 'Brasil Brasil Brasil'

A Linguagem Python provê dois comandos para obter documentação a respeito de objetos definidos no escopo atual da memória. Podemos, por exemplo, chamar o comando **help** passando um objeto qualquer como parâmetro:

1. >>> help(objeto)

Objetos

Em Python, qualquer coisa é um objeto: uma classe, um método, uma função, um texto ou um número. Qualquer tipo de dado, como *int* e *string*, por exemplo, também é objeto e se comporta como objeto.

Isso quer dizer que qualquer coisa em Python pode ser passada como parâmetro para uma função ou método, pois qualquer coisa é um objeto, e também quer dizer que qualquer objeto pode conter atributos e métodos (mas veremos isso mais adiante).

Vamos, por exemplo, utilizar o comando **help** passando como parâmetro o número inteiro **1**. **1** é um objeto do tipo **int** (inteiro); quando passarmos **1** para a função **help** teremos como resposta uma descrição detalhada a respeito da classe **int** e todos os seus métodos. Levando em conta que **1** é uma instância da classe **int**, podemos então utilizar qualquer método de **int** em **1**.

■ Veja o resultado:

```
1. >>> help(1)
2. Help on int object:
3.
4. class int(object)
5. | int(x[, base]) -> integer
6. |
7. | Convert a string or number to an integer, if possible. A floating point
8. | argument will be truncated towards zero (this does not include a string
9. | representation of a floating point number!) When converting a string, use
10. | the optional base. It is an error to supply a base when converting a
11. | non-string. If base is zero, the proper base is guessed based on the
12. | string content. If the argument is outside the integer range a
13. | long object will be returned instead.
14. |
15. | Methods defined here:
16. |
17. | __abs__(...)
18. | x.__abs__() <==> abs(x)
19. : (continua.....)
```

Perceba que aqui só mostramos uma pequena parte do retorno de **help**, pois a lista de métodos é muito grande. O mesmo resultado pode ser obtido utilizando a própria classe **int**:

```
1. >>> help(int)
```

help exibe uma documentação muito detalhada, e às vezes queremos ver apenas uma listagem de métodos disponíveis em um certo objeto - para isso, utilizamos o comando **dir**.

Perceba que **dir** e **help** também são objetos, são funções que aqui chamaremos de **comandos**, apenas para diferenciar as funções da linguagem Python das funções de um programa específico, criado por você ou não.

Tente executar `help(dir)` ou `dir(help)` e veja o resultado:

```
1. >>> dir(1)
2. ['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__',
3.  '__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '__format__',
4.  '__getattr__', '__getnewargs__', '__hash__', '__hex__', '__index__', '__init__',
5.  '__int__', '__invert__', '__long__', '__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
6.  '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
7.  '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
8.  '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__',
9.  '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
10. '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'conjugate', 'denominator',
11. 'imag', 'numerator', 'real']
```

Acima estão listados todos os métodos disponíveis na classe **int** que podem ser chamados a partir do objeto **1**.

Veja como invocar um método a partir de um objeto:

```
1. >>> objeto = 234 #Criamos uma variavel(objeto) e atribuímos o valor inteiro 234
2. >>> objeto.__hex__() #O método __hex__() da classe **int** converte para hexadecimal
3. '0xea'
```

Curiosidade, saiba mais sobre a filosofia Python

digite o seguinte código no terminal interativo:

```
1. >>> import this
```

Tipos dinâmicos

O Python efetua tipagem dinâmica dos objetos, uma variável não precisa possuir um tipo específico de dados, e as variáveis também não precisam ser declaradas.

Considere uma variável Python como sendo um espaço em memória que pode receber referência para objetos com valores de qualquer tipo. Portanto, são os valores que possuem tipo, e este tipo é definido dinamicamente.

Podemos verificar qual é o tipo de dado de um valor utilizando o comando **type**:

```
1. >>>type(1)
2. <type 'int'>
3. >>>type(1.25)
4. <type 'float'>
5. >>>type('texto')
6. <type 'str'>
7. >>>variavel = 'Mensagem'
8. >>>type(variavel)
9. <type 'str'>
```

str (strings)

Python suporta o uso de dois tipos de string: ASCII e Unicode.

ASCII são as strings usuais delimitadas por `'...'`, `"..."` ou por `"""..."""`. Três aspas delimitam strings com múltiplas linhas. Strings Unicode devem começar com uma letra **u**, seguida da string.

Unicode pode ser convertido para ASCII e vice-versa. Por padrão, o web2py utiliza UTF8 para strings.

Interpolação de variáveis em strings

É possível interpolar o conteúdo de variáveis em strings de diversas maneiras:

1. `>>> 'estamos no ano de ' + str(2010)`
2. `estamos no ano de 2010`
3. `>>> 'estamos no ano de %s ' % (2010)`
4. `estamos no ano de 2010`
5. `>>> 'estamos no ano de %(ano)s ' % dict(ano=2010) # recomendado`
6. `estamos no ano de 2010`

A última opção é mais explícita, e causa menos erros. É a maneira recomendada sempre que possível.

A partir da versão 2.6, está disponível outra maneira de interpolação, utilizando a função `format()`

1. `>>> 'Desenvolvimento web com {0} e {1}'.format('Python','web2py')`
2. `'Desenvolvimento web com Python e web2py'`
3. `>>> 'Desenvolvimento web com {linguagem} e {framework}'\`
4. `... .format(framework='web2py',linguagem='Python')`
5. `'Desenvolvimento web com Python e web2py'`

Estruturas de dados

Listas

Listas são coleções heterogêneas de objetos, que podem ser de qualquer tipo, inclusive outras listas.

As listas no Python são mutáveis, podendo ser alteradas a qualquer momento. Elementos de uma lista podem ser acessados através de seu índice.

Sintaxe

1. `lista = [a, b, ..., z]`

Operações com listas

Listas são objetos iteráveis, ordenados e indexáveis. Veja os seguintes exemplos:

```
1. >>> linguagens = ['Python','PHP','Ruby','Java','Lua'] #criando a lista
2. >>> linguagens.append('VB') #Adicionando um item à lista
3. >>> for linguagem in linguagens: print linguagem
4. ...
5. Python
6. PHP
7. Ruby
8. Java
9. Lua
10. VB
11. >>> linguagens[-1] #o último elemento da lista
12. 'VB'
13. >>> linguagens[0] #o primeiro elemento da lista
14. 'Python'
15. >>> linguagens[2:]#tudo do terceiro item em diante
16. ['Ruby', 'Java', 'Lua', 'VB']
17. >>> linguagens[2:5]#do terceiro até o quinto elemento
18. ['Ruby', 'Java', 'Lua']
19. >>> linguagens.remove('VB') #Removendo
20. >>> linguagens.sort() #Ordenando
21. >>> linguagens
22. ['Java', 'Lua', 'PHP', 'Python', 'Ruby']
23. >>>
24.
```

Tuplas

Tuplas são semelhantes às listas, porém são imutáveis: não se pode acrescentar, apagar ou fazer atribuições aos itens.

Sintaxe:

```
1. >>> minhaturpla = (a, b, ..., z) # parênteses opcionais
2. >>> minhaturpla = a, b, ..., z # o resultado é o mesmo
3. >>> minhaturple = ( 1, ) # Tuplas com apenas um item terminam com uma vírgula
```

As tuplas são muito úteis quando necessitamos garantir a integridade dos dados, ao criar entradas de configuração que não se alteram durante o programa, por exemplo.

Listas podem ser convertidas em tuplas, e tuplas podem ser convertidas em listas:

```
1. >>> minhalista = list(minhatupla)
2. >>> minhaturpla = tuple(minhalista)
```

Embora as tuplas possam conter elementos mutáveis, não é possível fazer atribuições, pois isso modificaria a referência ao objeto.

```
1. >>> minhaLista = [1,2]
2. >>> minhaTupla = (minhaLista,4,[1,2],'Python')
3. >>> minhaTupla
4. ([1, 2], 4, [1, 2], 'Python')
5. >>> minhaTupla[0].append(3) #adicionamos um item a minhaLista
6. >>> minhaTupla
7. ([1, 2, 3], 4, [1, 2], 'Python')
8. >>> minhaTupla[0] = [1,2,3,4]
9. Traceback (most recent call last):
10. File "<stdin>", line 1, in <module>
11. TypeError: 'tuple' object does not support item assignment
```

As tuplas são mais eficientes do que as listas, pois consomem menos recursos e demandam menos operações computacionais.

Tuplas são muito utilizadas para empacotar e desempacotar valores:

```
1. >>> a = 2, 3, 'hello'
2. >>> x, y, z = a
3. >>> print x
4. 2
5. >>> print z
6. hello
```

Dicionários

Um dicionário é uma lista de associações compostas por uma chave única e estruturas correspondentes. Dicionários são mutáveis, tais como as listas.

Os dados nos dicionários são armazenados em uma estrutura chave:valor

```
1. >>> pessoa = {'nome':'João','idade':35,'altura':1.69}
```

Um dicionário é uma estrutura de dados que se parece com um banco de dados, e seus valores podem ser acessados através de sua chave.

```
1. >>> pessoa['nome']
2. 'João'
3. >>> pessoa['idade']
4. 35
5. >>> Pessoa['Peso'] = 67
```

Podemos colocar dicionários dentro de dicionários:

```
1. >>> pessoas = {}
2. >>> pessoas['Bruno'] = {'idade':27,'Cidade':'Cotia'}
3. >>> pessoas['Claudia'] = {'idade':29,'Cidade':'Taubate'}
4. >>> pessoas
5. {'Claudia': {'idade': 29, 'Cidade': 'Taubate'}, 'Bruno': {'idade': 27, 'Cidade': 'Cotia'}}
6. >>> pessoas['Claudia']
7. {'idade': 29, 'Cidade': 'Taubate'}
8. >>> pessoas['Claudia']['idade']
9. 29
```

Dicionários são iteráveis, podemos percorrer seus elementos:

```
1. >>> for pessoa in pessoas: print pessoa + ' ' + pessoas[pessoa]['Cidade']\
2. ... + ' ' + str(pessoas[pessoa]['idade'])
3. ...
4. Claudia Taubate 29
5. Bruno Cotia 27
```

No caso acima o objeto *pessoas* é um dicionário, e possui dentro dele outros dois dicionários identificados pelas chaves 'Bruno' e 'Claudia'. Esses dois dicionários, por sua vez, também armazenam um dicionário dentro de si, cada um com as chaves 'idade' e 'cidade'. Poderíamos ir mais longe e incluir inúmeros dicionários aninhados.

As chaves nos dicionários podem ser dos tipos *int*, *string* ou qualquer objeto que implemente o método `__hash__`. Valores podem ser de qualquer tipo, podemos ter diferentes tipos de chaves e valores em um único dicionário. Se os valores forem todos alfanuméricos, o dicionário pode ser declarado com a sintaxe alternativa:

```
1. >>> pessoas = dict(Bruno={'idade':27,'Cidade':'Cotia'},Claudia={'idade':29,'Cidade':'Taubate'})
2. >>> pessoas
3. {'Claudia': {'idade': 29, 'Cidade': 'Taubate'}, 'Bruno': {'idade': 27, 'Cidade': 'Cotia'}}
4. >>> pessoas['Claudia']['idade']
5. 29
```

Os métodos mais utilizados em dicionários são *has_key*, *keys*, *values* and *items*:

```
1. >>> pessoas.keys()
2. ['Claudia', 'Bruno']
3. >>> pessoas.values()
4. [{ 'idade': 29, 'Cidade': 'Taubate' }, { 'idade': 27, 'Cidade': 'Cotia' }]
5. >>> pessoas.items()
6. [('Claudia', { 'idade': 29, 'Cidade': 'Taubate' }), ('Bruno', { 'idade': 27, 'Cidade': 'Cotia' })]
```

O método *items* produz uma lista de tuplas, cada uma contendo a chave e seu respectivo valor. Esta saída é muito útil para efetuar operações como esta:

```
1. >>> for item in pessoas.items():
2. ...     print item
3. ...
4. ('Claudia', { 'idade': 29, 'Cidade': 'Taubate' })
5. ('Bruno', { 'idade': 27, 'Cidade': 'Cotia' })
```

Itens de listas e dicionários podem ser excluídos com o comando *del*:

```
1. >>> del pessoas['Bruno']
2. >>> pessoas
3. {'Claudia': { 'idade': 29, 'Cidade': 'Taubate' }}
```

Internamente, o Python usa o operador hash para converter objetos para *int*, e utiliza este valor inteiro para determinar onde armazenar o valor.

```
1. >>> hash("hello world")
2. -1500746465
```

Sobre endentação

Dentro da computação, Endentação (reco, neologismo derivado da palavra em inglês indentation, também encontram-se as formas indentação e indentação) é um termo aplicado ao código fonte de um programa para indicar que os elementos hierarquicamente dispostos têm o mesmo avanço relativamente à posição (x,0).

Na maioria das linguagens a indentação tem um papel meramente estético, tornando a leitura do código fonte muito mais fácil (read-friendly), porém é obrigatória em outras. Python, occam e Haskell, por exemplo, utilizam-se desse recurso tornando desnecessário o uso de certos identificadores de blocos ("begin" e/ou "end").

Fonte:wikipedia

Python usa a endentação para delimitar blocos de código. Um bloco inicia-se com uma linha terminada em dois pontos (:), e continua para todas as linhas que tenham uma endentação similar ou mais recuada. Por exemplo:

```
1. >>> i = 0
2. >>> while i < 3:
3. >>>     print i
4. >>>     i = i + 1
5. >>>
6. 0
7. 1
8. 2
```

É comum utilizar 4 espaços para cada nível de endentação. É uma boa prática não misturar espaços com tabulações, pois pode gerar problemas.

Iteração de elementos

Em Python você pode efetuar *loops* em objetos iteráveis utilizando os comandos **for...in** e **while**:

```
1. >>> lista = [0, 1, 'hello', 'python']
2. >>> for item in lista:
3. ...     print item
4. ...
5. 0
6. 1
7. hello
8. python
```

Duas funções que são bastante úteis: **xrange** e a **range**

```
1. >>> for i in xrange(3,6):
2. ...     print i
3. ...
4. 3
5. 4
6. 5
```

```
1. >>> for i in range(3):
2. ...     print i
3. ...
4. 1
5. 2
6. 3
```

Esta sintaxe é equivalente ao seguinte código C/C++/C#/Java:

```
1. for(int i=0; i<3; i=i+1) {print(i);}
```

Você pode abortar um *loop* utilizando o comando **break**

```
1. >>> for i in [1, 2, 3]:
2. ...     print i
3. ...     break
4. 1
```

É possível ir para a próxima iteração, sem a necessidade de executar todo o código definido no bloco, utilizando o comando **continue**

```
1. >>> for i in [1, 2, 3]:
2. ...     print i
3. ...     if i==2:
4. ...         continue
5. ...     print 'test'
6. ...
7. 1
8. test
9. 2
10. 3
11. test
```

while

while (enquanto) em Python funciona de maneira muito similar a outras linguagens. Este comando efetua a repetição indefinidas vezes, testando uma condição no final de cada iteração até que a condição seja igual a **False**

```
1. >>> while i < 10:
2. ...     i = i + 1
3. ...
4. >>> print i
5. 10
```


Expressões condicionais

O uso de condições em Python é intuitivo:

```
1. >>> for i in range(3):
2. ...     if i == 0:
3. ...         print 'Zero'
4. ...     elif i == 1:
5. ...         print 'Um'
6. ...     else:
7. ...         print 'Outro'
8. ...
9. Zero
10. Um
11. Outro
```

elif é o mesmo que **else if**, e você pode ter quantos **elifs** desejar, mas só é permitido apenas um **else** sempre como última condição do bloco.

Funções

Funções são blocos de código nomeados. Criadas para permitir a reutilização de código, as funções podem receber argumentos. Algumas funções já estão presentes no interpretador Python.

Exemplo de função

```
1. >>> def nome_da_funcao(argumento1, argumento2):
2. ...     bloco de código
3. ...     return <resultado>
```

O web2py segue o estilo de marcação determinado no [PEP-8](http://www.python.org/dev/peps/pep-008/), por isso será comum ver funções nomeadas com o uso de 'underline', ex: `nome_da_funcao`. Porém, você pode preferir utilizar outro estilo, como camelCasing ou PascalCasing na hora de criar as suas próprias funções.

Retorno de valores

Se a função retorna algum valor, como no caso de uma soma ou outro tipo de tratamento de dado, é utilizado o comando **return** para especificar o que será retornado.

```
1. >>> def soma(x,y):
2. ...     return x + Y
3. ...
4. >>> soma(1,2)
5. 3
```

Uma única função poderá retornar mais de um objeto, e uma função pode receber argumentos nomeados:

```
1. >>> def consulta_produto(id,desconto=False):
2. ...     if id:
3. ...         produto = id
4. ...         if desconto==False:
5. ...             return produto,'sem desconto'
6. ...         else:
7. ...             return produto,'com desconto'
8. ...
9. >>> consulta_produto(desconto=True,id=10)
10. (10, 'com desconto')
```

lambda

Expressões **lambda** são funções que não possuem nome e que são criadas dinamicamente:

```
1. >>> def soma(a, b):
2. ...     return a+b
3. ...
4. >>> soma_alterada = lambda a: 5 + soma(a, 3)
5. >>> soma_alterada(2)
6. 10
7. >>>
```

A expressão **lambda [a]:[b]** pode ser lida como: uma função que recebe o argumento **a** e retorna o valor **b**. Mesmo sendo uma função sem nome, ela pode ser referenciada por uma variável **soma_alterada** e desta maneira é possível invocá-la. **lambda** é muito útil para efetuação de refatoramento e sobrecarga (*overloading*) de métodos e funções.

Executando código pré-determinado

Diferente de C#, Java ou PHP, Python é uma linguagem totalmente interpretada. Isso significa que Python tem a habilidade para executar blocos de código armazenados em strings:

```
1. >>> a = "print 'web2py Brasil'"
2. >>> exec(a)
3. 'web2pyBrasil'
4.
```

Comentários e docstrings

Existem duas maneiras para inserir um comentário em Python. Para comentários gerais no meio do bloco de código, basta utilizar o caractere #

```
1. >>> def funcao(): # aqui definimos uma função
2. ...     # na linha abaixo retornamos o valor
3. ...     return <valor>
```

Também podemos utilizar docstrings, que possuem duas funcionalidades. Docstring é uma maneira de inserir comentários, ou string multilinhas, mas também pode ser

{{ Desenvolvimento web com Python e web2py }}

utilizada para gerar documentação automaticamente. Docstrings são definidas com três aspas simples ou duplas.

```
1. >>> def funcao():
2. ...     """
3. ...     Aqui começa minha docstring, este texto servirá
4. ...     como documentação que pode ser gerada por ferramentas como **epydoc**
5. ...     docstrings também são usadas por ferramentas de TDD como **doctests**
6. ...     geralmente explicam como utilizar a função, ou incluem código python
7. ...     >>> funcao()
8. ...     'retorno da funcao'
9. ...     """
10. ...     return 'retorno da funcao'
```

Você pode acessar o texto da docstring de uma função, ou até mesmo executar código Python contido na docstring.

```
1. >>> def funcao():
2. ...     """
3. ...     docstring da funcao
4. ...     essa funcao nao tem retorno
5. ...     """
6. ...     return None
7. ...
8. >>> funcao()
9. >>> funcao.__doc__
10. '\n docstring da funcao\n  essa funcao nao tem retorno\n  '
```

O mesmo resultado é obtido também com o uso do comando **help(funcao)**.

Módulos

Cada arquivo em Python é chamado de módulo. Módulos são conjuntos de códigos como funções, classes, métodos, variáveis etc. Vários módulos podem se comunicar através do comando **import modulo**.

Vamos pegar o seguinte módulo como exemplo:

conteúdo do arquivo modulox.py

```
1. def soma(a,b):
2.     return a + b
```

Agora podemos utilizar a função de soma definida em modulox.py em qualquer parte do nosso projeto

importando módulos

Existem dois modos principais para importar um módulo:

- **Importando com alto acoplamento:**
desta maneira as funções e variáveis do **modulox** precisam ser acessadas mantendo-se o nome do módulo como em **modulox.soma()**

```
1. >>> import modulox      # aqui importamos o módulo **modulox.py**
2. >>> print modulox.soma(3,6)  # aqui invocamos a função **soma**
3. 9
```

- **Importando diretamente:**

desta forma todos os objetos definidos em **modulox** ficam diretamente disponíveis para acesso dentro do programa atual, e ainda temos a opção de escolher quais objetos iremos importar.

Para importar todos os objetos utilizaremos *

1. `>>> from modulox import * # todos os objetos definidos em modulox.py`
2. `>>> from modulox import soma # apenas a função soma`
3. `>>> from modulox import soma as minhasoma # importar a função soma e renomear`

Em todos os casos acima, os objetos importados ficam diretamente acessíveis:

1. `>>> print soma(3,6)`
2. `9`
3. `>>> print minhasoma(3,6)`
4. `9`

List Comprehensions

Compreensão de listas é uma construção que permite processar listas de uma maneira muito similar à linguagem matemática.

A sintaxe é [**<expressão>** for **<variável>** in **<lista>** if **<condicao>**]

onde: **<variável>** in **<lista>** pode se repetir N vezes.

e a **<condição>** é opcional.

Vamos utilizar a função soma, por exemplo:

1. `>>> print [modulox.soma(numero,5) for numero in range(10) if numero!=5]`
2. `[5, 6, 7, 8, 9, 11, 12, 13, 14]`

No exemplo acima, executamos a função soma para somarmos **5** a cada **número** da lista gerada pelo comando **range(10)** apenas se o **numero** for diferente de **5**.

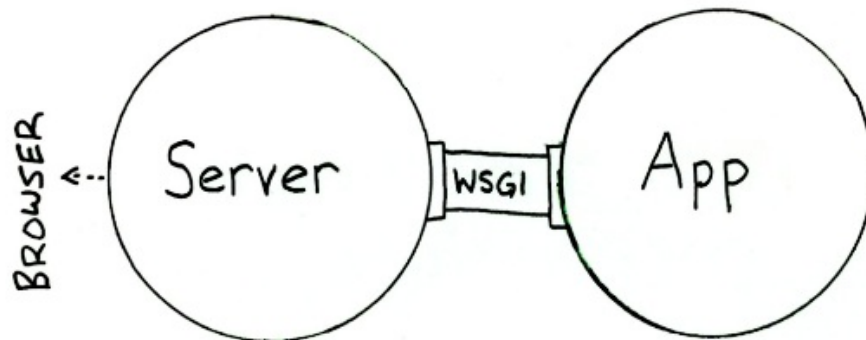
Referências

[1] ["www.python.org/dev/peps"](http://www.python.org/dev/peps/)

[2] ["http://www.python.org.br/wiki/InicieSe"](http://www.python.org.br/wiki/InicieSe)

WSGI

Também pronunciado como *uisgui*, WSGI é a especificação que determina os padrões de acesso e comunicação entre servidores web e aplicações escritas em Python. WSGI é descrito em detalhes no [PEP 333](#)



WSGI **não** foi feito para desenvolvimento de aplicativos, mas sim para desenvolvimento de frameworks e webservers.

Exemplo de uma aplicação compatível com WSGI:

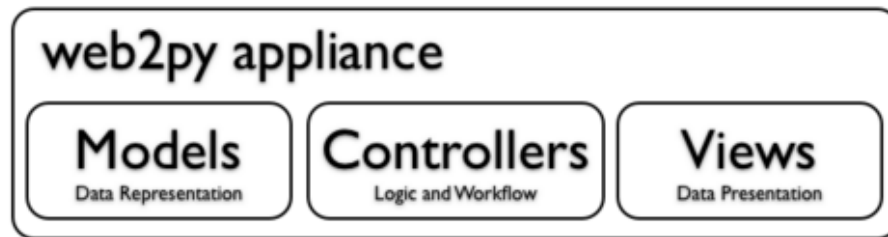
```
1. def app(environ, start_response):
2.     start_response('200 OK', [('Content-Type', 'text/plain')])
3.     from modulos import soma
4.     resultado = soma(2,3)
5.     yield 'Resultado da soma: %s' % resultado
```

Yield, em Python, é usado nos generators. Generators são uma forma de gerar iterators. O yield faz com que a função pare onde ele for chamado, e que na próxima vez em que for chamado, os valores sejam os mesmos da vez anterior.

Caso você queira escrever seu próprio framework, ou desenvolver um webserver, aconselho a leitura do site oficial do projeto [WSGI](#)

O Padrão MVC

O padrão de MVC (Model-View-Controller) possibilita uma separação inteligente entre o modelo de dados (Model), a lógica da aplicação (Controller) e a interface de apresentação (View).



Modelos (model)

É o local onde definimos e gerenciamos os modelos de dados da aplicação, efetuamos conexões com bancos de dados e definimos a modelagem das tabelas, constantes, variáveis e configurações de acesso global.

Controladores (controller)

São as ações da aplicação, onde definimos as regras de negócio e as validações de tempo de execução. O controller é quem recebe a entrada de dados, invoca os objetos do modelo de dados, efetua as validações e envia como resposta uma visão.

Visão (view)

As visões apresentam os dados do model, invocados e tratados pelo controller. Podem ser qualquer tipo de interface com usuários (páginas HTML, Feeds RSS etc) ou com outros sistemas (XML webservice, API REST etc).

Referências

[1] ["http://www.python.org/dev/peps"](http://www.python.org/dev/peps)

[2] ["http://wsgi.org/wsgi"](http://wsgi.org/wsgi)

Preparando o ambiente de desenvolvimento

O web2py possibilita que o desenvolvedor se dedique integralmente à criação da aplicação, pois não demanda instalações, nem configurações complicadas: basta baixar o framework e começar a desenvolver.

É multiplataforma, ou seja, você pode rodá-lo no Windows ou no Mac OS, utilizando os executáveis disponíveis para download no site oficial <http://www.web2py.com> (para quem utiliza os sistemas operacionais citados, basta baixar e executar).

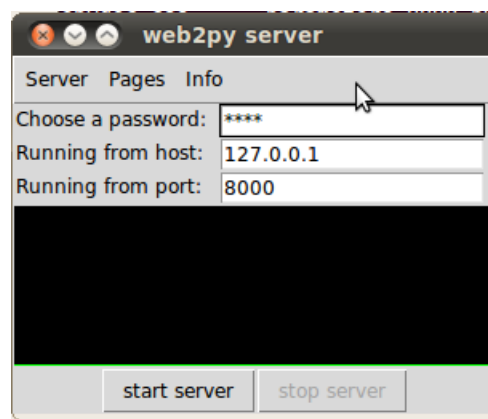
Se você estiver utilizando Linux, baixe a versão código fonte. Se o seu Linux for o Ubuntu, pode ser que seja necessário instalar a biblioteca python-tk, através da utilização do seguinte comando:

1. `sudo apt-get install python-tk`

Faça download da versão código fonte do web2py e descompacte-o, então, dentro do diretório “web2py”, abra um terminal e execute o comando:

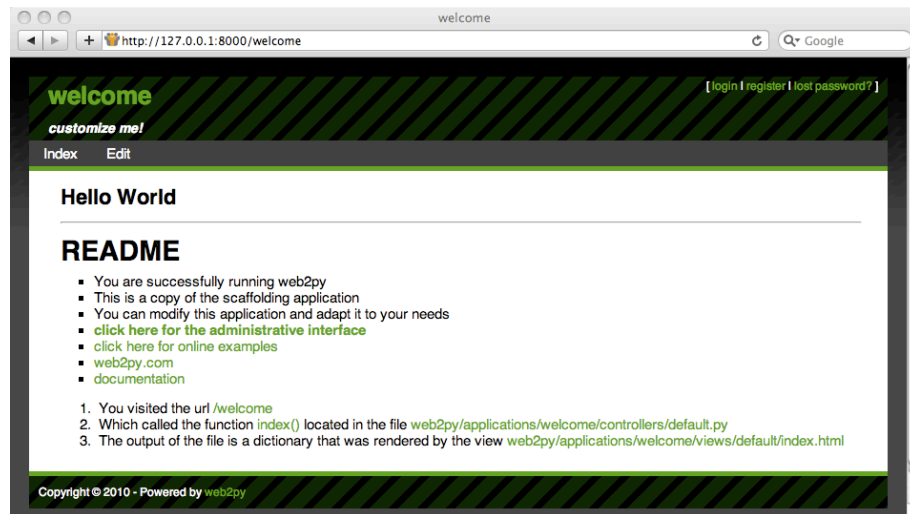
1. `python web2py.py`

É possível rodar o web2py em um servidor com Apache, ou em qualquer outro servidor com suporte para WSGI, CGI/FASTCGI ou com função de proxy. O web2py também pode se conectar a vários tipos de bancos de dados, porém, como o framework já vem com um webserver próprio e com os pacotes necessários para acesso ao SQLite, utilizaremos a configuração padrão. Ao rodar o comando citado acima, a seguinte tela será exibida:



Defina uma senha para acesso à aplicação de administração e clique em Start Server. Seu navegador abrirá automaticamente, exibindo a aplicação welcome (aplicação modelo). Além disso, temos a aplicação admin, uma interface de administração para as outras aplicações, e também a aplicação examples, que apresenta diversos exemplos de código.

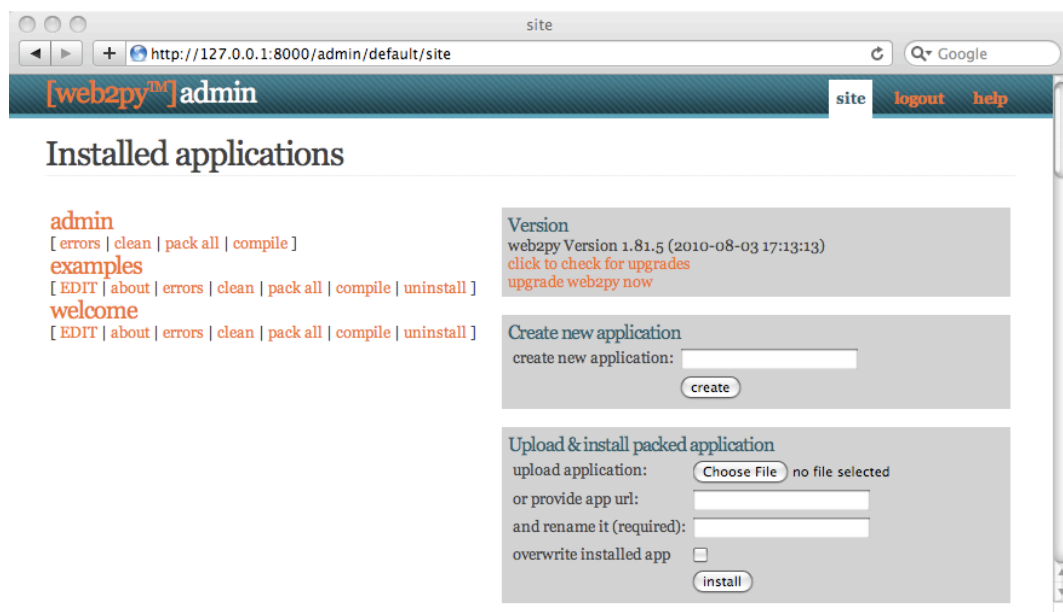
A aplicação welcome:



Clique em [click here for admin...] e entre com a senha definida no passo anterior.

admin - A interface de administração:

admin é o nome da interface de administração que o web2py fornece. É uma aplicação como qualquer outra, porém com poderes de administrar os arquivos e bancos de dados das demais aplicações.



Esta página exibe todas as aplicações instaladas e permite que o administrador as gerencie.

Através desta página de administração você pode executar as seguintes operações:

Install - Instalar uma nova aplicação em seu web2py. É possível instalar uma aplicação informando sua URL ou o caminho para um pacote w2p. Aplicativos w2p totalmente funcionais estão disponíveis em <http://web2py.com/appliances>.

uninstall - Desinstala uma aplicação.

create - Cria uma nova aplicação

Quando você cria uma nova aplicação, o web2py cria um clone da aplicação de modelo **welcome**, incluindo um arquivo de modelo em *models/db.py* que cria e efetua a conexão com um banco de dados **SQLite**, instancia e configura as classes de autenticação **Auth**, de criação de formulários CRUD e de Serviços (XML-RPC). Ele também cria um controller *controllers/default.py* que expõe as ações *index*, *download*, *user* para gerenciamento de usuários, e *call* para chamada de serviços.

package - Permite empacotar uma aplicação que você desenvolveu e distribuí-la no formato w2p.

compile - Compila uma aplicação para a distribuição binária sem o código fonte.

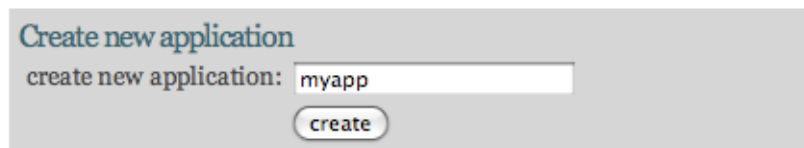
errors - Exibe os tickets de erro gerados pela aplicação.

clean up - Limpa todos os arquivos temporários, sessões, tickets de erro e arquivos de cache de uma aplicação.

EDIT - Abre a aplicação em modo de edição.

Criando sua primeira aplicação

Comece criando uma aplicação chamada *myapp*:



Create new application
create new application:

Após você clicar em [create], a aplicação será criada como uma cópia da aplicação **welcome**.

Installed applications

admin

[errors | clean | pack all | compile]

examples

[EDIT | about | errors | clean | pack all | compile | uninstall]

myapp

[EDIT | about | errors | clean | pack all | compile | uninstall]

welcome

[EDIT | about | errors | clean | pack all | compile | uninstall]

Version

web2py Version 1.10.1
click to check for updates
upgrade web2py now

Create new application
create new application

Upload & install application
upload application
or provide app url

Para executar sua nova aplicação visite a seguinte URL:

<http://127.0.0.1:8000/myapp>

Você verá uma cópia da aplicação **welcome**, clique em EDIT para entrar em modo de edição da aplicação:



A página de edição exibe o que existe dentro da aplicação. Toda aplicação web2py consiste em um certo conjunto de arquivos; os mais importantes estão nas seguintes categorias:

[models]

Modelos – É o local onde definimos e gerenciamos os modelos de dados da aplicação, efetuamos conexões com bancos de dados e definimos a modelagem das tabelas, constantes, variáveis e configurações de acesso global.

[controllers]

Controladores – São as ações da aplicação. Aqui definimos as regras de negócio e as validações de tempo de execução; o controller é quem recebe a entrada de dados, invoca os objetos do modelo de dados, efetua as validações e envia como resposta uma visão.

[views]

Visões – São as visões que servirão para apresentar os dados do model, invocados e tratados pelo controller. São criadas a partir de templates que podem responder conteúdo no formato HTML, RSS, XML e JSON, entre outros.

[languages]

Linguagens – Local onde definimos arquivos de linguagem que permitem a internacionalização das aplicações.

[static]

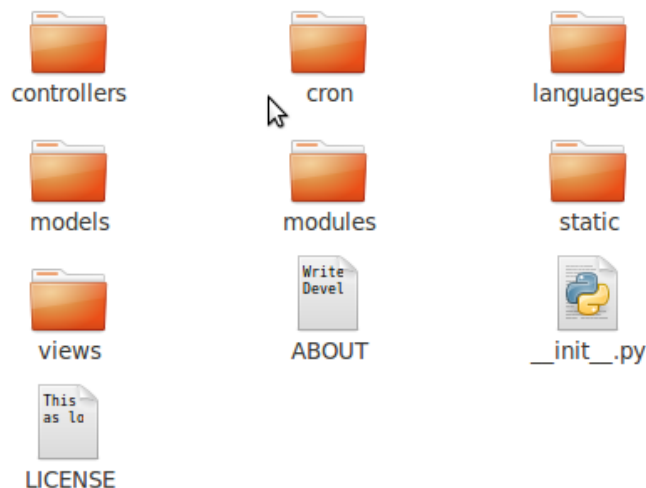
Estáticos – Neste diretório, inserimos arquivos que não necessitam de processamento, como estruturas estáticas de layout, imagens, arquivos de estilo CSS e JavaScript.

[modules]

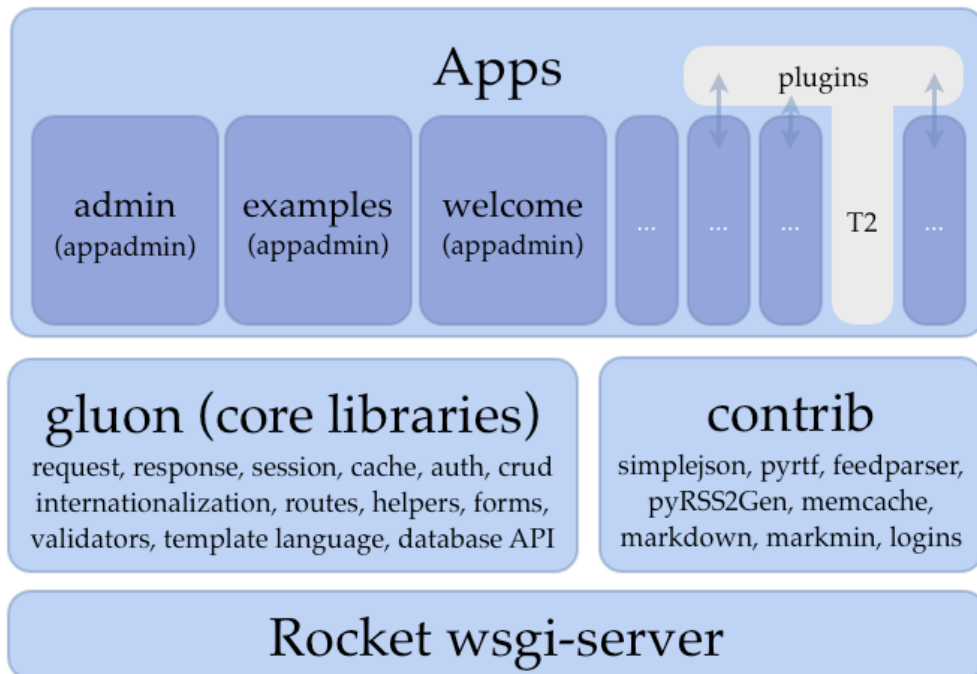
Módulos - Aqui colocaremos bibliotecas e módulos Python que não fazem parte do web2py, mas que podem ser importados e utilizados nas aplicações.

Cada arquivo ou pasta exibido na interface administrativa, corresponde a um arquivo físico localizado no diretório da aplicação. Qualquer operação realizada na interface administrativa pode ser realizada através de comandos de terminal, e os arquivos podem ser editados em qualquer editor de textos ou IDE.

As aplicações ainda possuem outros tipos de arquivo (bancos de dados, arquivos de sessão, tickets de erro etc.), mas estes arquivos não são listados na página de edição, pois não são criados ou gerenciados pelo desenvolvedor: eles são criados e modificados pela própria aplicação.

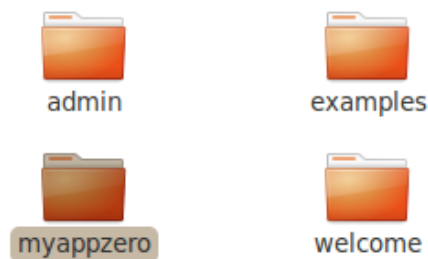


Camadas, módulos e pacotes do web2py



A partir do zero

Vamos criar uma aplicação a partir do zero. Abra a pasta **web2py/applications** e dentro dela crie uma nova pasta chamada **myappzero**.



Dentro desta pasta criaremos os arquivos essenciais para que esta pasta se transforme em uma aplicação **web2py**.

__init__.py

Este arquivo informa ao interpretador que esta pasta é um pacote Python e desta forma poderá ser executado como aplicação web2py. Este arquivo é nomeado com dois underlines, no início e no final, **__init__.py**, e não possui conteúdo.

ABOUT

Este arquivo conterá a descrição de sua aplicação. Ele é um arquivo obrigatório, apesar de ser permitido que fique em branco. Recomenda-se que seja preenchido pelo menos com as informações básicas: descrição da aplicação, versão e nome do autor. É interessante informar, neste arquivo, alguma dependência que a aplicação possa ter, e incluir instruções de instalação e configuração.

LICENSE

Este arquivo conterá a declaração de licença para o software desenvolvido com web2py. A licença do web2py permite que a aplicação seja distribuída de forma comercial, desde que tenha seu próprio arquivo de licença.

controllers

Nesta pasta criaremos os arquivos que conterão a lógica principal da aplicação. Em cada um destes arquivos criaremos as funções Python que chamaremos de **ações**, e cada ação será mapeada para uma URL:

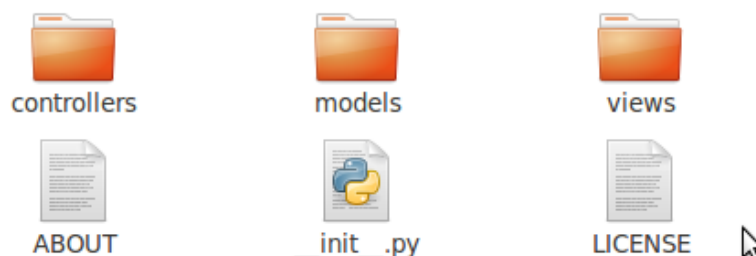
<http://servidor/aplicação/controller/ação/argumento1/argumento2>

models

Nesta pasta criaremos os modelos de dados e as configurações globais. Os arquivos dentro de **models** serão executados em ordem alfabética.

views

Nesta pasta criaremos os arquivos de layout e as visões para a apresentação dos dados. O conteúdo da pasta **myappzero**, passará a ser uma aplicação web2py.

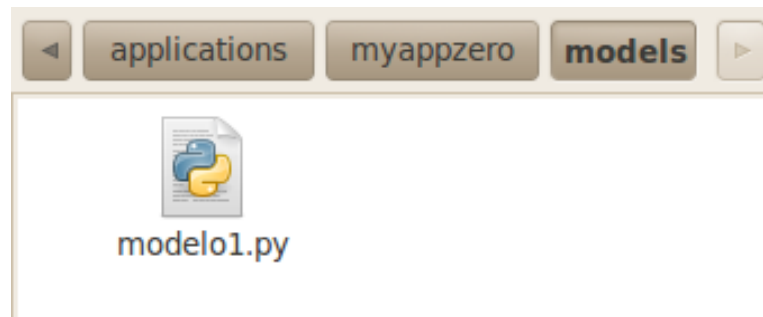


Modelos (models)

Vamos começar pelos modelos. De acordo com a especificação do padrão MVC, os modelos devem tratar apenas das estruturas e modelos de dados, porém o web2py dá ao desenvolvedor a flexibilidade de escrever qualquer tipo de código Python dentro do nível dos modelos.

O web2py lê os arquivos de modelo em ordem alfabética e executa todo código contido nestes arquivos; instância objetos e variáveis de acesso global; executa funções; conecta, modela, verifica e cria tabelas no banco de dados.

Para entender o ciclo de vida de uma aplicação web2py, crie um arquivo chamado **modelo1.py** dentro da pasta **applications/myappzero/models**:



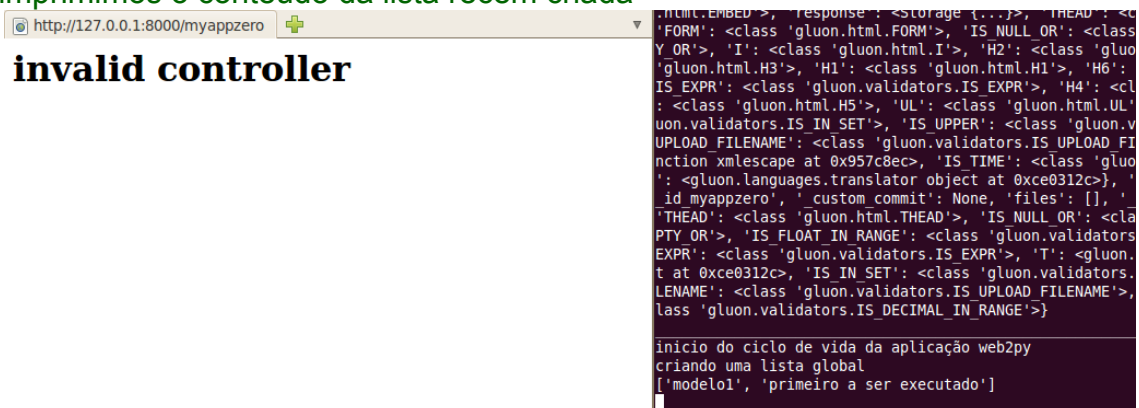
```
1. # -*- coding: utf-8 -*-
2. print globals()
3. print '_'*80
4. print 'início do ciclo de vida da aplicação web2py'
5. print 'criando uma lista global'
6. lista1 = ['modelo1', 'primeiro a ser executado']
7. print lista1
```

Quando incluímos o comando **print** em qualquer arquivo de uma aplicação web2py, esta informação é enviada para o **terminal** onde o web2py está sendo executado. Este é um recurso muito útil para depurar a aplicação.

Abra o terminal de execução do web2py e o deixe visível; ao lado abra a janela do navegador e aponte para a URL <http://127.0.0.1:8000/myappzero>

A saída de dados do modelo **modelo1.py** no terminal será:

- `print globals()` - Exibe todos os objetos que estão no escopo global do interpretador Python, incluindo os objetos carregados pelo próprio Python e também os objetos carregados pelo web2py.
- `print '_'*80` - Apenas para dividir o conteúdo na tela, imprimimos 80 vezes o caractere *underline*
- exibimos algumas mensagens de status
- criamos um objeto do tipo lista contendo dois elementos; esta lista a partir de agora ficará no escopo global
- imprimimos o conteúdo da lista recém criada



O web2py mostrará a mensagem **invalid controller**, pois estamos chamando diretamente a aplicação, que neste momento apenas invocará a execução dos modelos.

Todos os arquivos da pasta **models** serão executados em ordem alfabética e todos os objetos criados nestes arquivos serão instanciados na memória de escopo global. A memória poderá ser acessada através dos controladores e até mesmo através das visões.

Na janela de terminal onde o web2py está sendo executado, você verá a saída dos comandos `print` que escrevemos nos controllers.

Vamos agora nos certificar de que o web2py está mesmo respeitando a ordem alfabética na execução dos modelos e criar um novo arquivo, na pasta **models**, chamado **modelo2.py**

```
1. # -*- coding: utf-8 -*-
2. print '_*80
3. print 'segundo modelo a ser executado pelo web2py'
4. print 'criando outra lista global'
5. lista2 = ['modelo2', 'segundo a ser executado']
6. print lista2
7. print 'importando o módulo calendário e imprimindo os meses'
8. import calendar
9. meses = [month for month in calendar.month_name]
10. print meses
```

Atualize a tela do seu navegador e verifique a saída no terminal.

```
LOAD IN RANGE': <class 'gluon.validators.IS_FLOAT_IN_RANGE'>, 'IS_EXPR': <class 'gluon.validators.IS_EXPR'>, 'T': <gluon.languages.translator object at 0xce10f8c>, 'IS_IN_SET': <class 'gluon.validators.IS_IN_SET'>, 'IS_UPLOAD_FILENAME': <class 'gluon.validators.IS_UPLOAD_FILENAME'>, 'IS_DECIMAL_IN_RANGE': <class 'gluon.validators.IS_DECIMAL_IN_RANGE'>}
```

```
início do ciclo de vida da aplicação web2py
criando uma lista global
['modelo1', 'primeiro a ser executado']
```

```
segundo modelo a ser executado pelo web2py
criando outra lista global
['modelo2', 'segundo a ser executado']
importando o módulo calendário e imprimindo os meses
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']
```

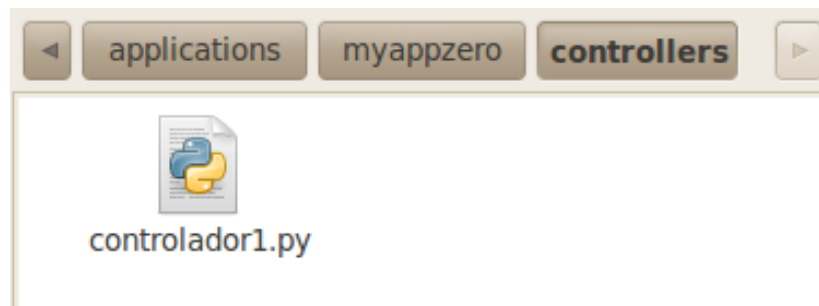
Nos modelos **modelo1** e **modelo2** criamos e instanciamos globalmente os objetos: **lista1**, **lista2** e **meses**. Estes objetos estão agora disponíveis para acessarmos através dos controladores e das views.

Verifique a pasta raiz da aplicação e veja que no momento em que efetuamos o primeiro request ao servidor, o web2py agora criou automaticamente todos os arquivos que faltavam na aplicação.

Controladores (controllers)

Agora vamos criar um arquivo controlador. Um controlador é o responsável pela execução de tarefas lógicas da aplicação, tratamento de dados e comunicação com as visões. Nos controladores criamos funções Python que chamamos de **ações** e cada ação poderá ser mapeada para uma URL. As ações podem responder para uma ou mais visões, assim como as visões poderão responder por uma ou mais ações. Os controladores têm acesso a todos os objetos que estão instanciados no escopo global da aplicação.

Dentro da pasta **controllers**, crie um arquivo chamado **controlador1.py**.



Edite o arquivo **controlador1.py** e defina uma função Python chamada **acao1**:

```
1. #-*- coding: utf-8 -*-
2. def acao1():
3.     print '*'*80
4.     print 'Inicio da execução da acao1 no controlador1'
5.     lista1.append('acao1 inseriu um item a lista1')
6.     print lista1
7.     del lista2[1]
8.     print 'acao1 excluiu um item de lista2'
9.     print lista2
10.    print 'acao1 irá se comunicar com a visão'
11.    return dict(lista1=lista1,lista2=lista2,meses=meses)
```

Atualize a tela do seu navegador e verifique a saída no terminal, repare que a mensagem que aparece no navegador agora é **invalid view**. Nosso último passo será criar a respectiva view.

O web2py expõe apenas as funções que não recebem parâmetros e que não começam com `_underline`; nós chamaremos estas funções de **ações**. As funções que recebem parâmetros e as começadas por `"_underline"` ficam disponíveis apenas através de outras funções (ações), middlewares ou serviços.

Mapeamento de URLs

O mapeamento de URLs (dispatching) no web2py é orientado a ações dos controladores. Cada ação de um controlador é acessível através de uma URL no seguinte formato:

<http://servidor:porta/aplicação/controlador/ação/argumentos?variáveis>

<http://servidor:porta>

- Endereço IP e porta do servidor, em desenvolvimento 127.0.0.1:8080

[/aplicação](#)

- Pasta da aplicação web2py, ao ser chamada executa o código dos modelos

[/controlador](#)

- Módulo controlador

[/ação](#)

- Ação de um controlador, ao ser chamada executa o código da função correspondente, recebe os parâmetros e variáveis

[/argumentos](#)

- Argumentos passados para o controlador são tratados como POST

[?variáveis](#)

- Variáveis são tratadas como GET (QueryString)

Para uma completa lista dos objetos de contexto visite a seguinte URL:

http://127.0.0.1:8000/examples/simple_examples/status

Visões (views)

Visões são modelos HTML, RSS, JSON, XML ou qualquer outro tipo de modelo para apresentação das informações. Por exemplo: tudo o que for apresentado ao navegador web faz parte da visão.

No web2py, o template das visões é marcado utilizando linguagem Python, e podemos fazer quase tudo que faríamos em um programa Python ou Controller diretamente dentro da View. Apesar de ser recomendado deixar a lógica toda no Controller, às vezes é muito útil colocar um pouco de inteligência nas Views.

O padrão para marcação segue os modelos abaixo, e os demais detalhes veremos no decorrer:

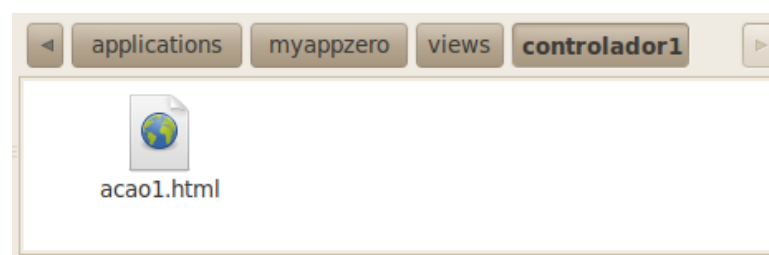
- O código da view não precisa seguir regras de endentação do Python
- A marcação para escape é feita dentro de {{ e }}
- Blocos de código começam nas linhas {{}} terminadas em :
- Blocos de código terminam onde encontram a instrução {{pass}}
- Nos casos em que a construção do bloco for clara, não será preciso usar o {{pass}}

Criando uma visão

As visões ficam no diretório **aplicação/views/**. Podemos dizer ao controlador qual arquivo de visão ele irá invocar, porém, caso não façamos isso, o web2py determinará o uso de um arquivo de visão padrão, que segue a mesma estrutura de nomenclatura e diretório.

Crie dentro da pasta **myappzero/views** uma pasta chamada **controlador1**. Quando uma ação do controlador1 for chamada, ela procurará dentro desta pasta uma view com o mesmo nome da ação, portanto:

Crie o arquivo **myappzero/views/controlador1/acao1.html**:



Este arquivo conterá o template para exibição dos dados da **acao1**, e poderemos executar qualquer código Python dentro dele. Todos os objetos que foram retornados pelos modelos e controlador estarão disponíveis para a visão:

```
1. <html>
2.   <head></head>
3.   <body>
4.     {{print ' '*80}}
5.     {{print "Visão exibida no navegador"}}
6.     {{=lista1}}
7.     <br />
8.     {{=lista2}}
9.     <br />
10.    <ul>
11.      {{for mes in meses:
12.        if mes:
13.          }}
14.          <li>{{=mes}}</li>
15.      {{ pass
16.      pass
17.      }}
18.    </ul>
19.  </body>
20. </html>
21.
```

Passagem de argumentos para a visão

O objeto **request** é o transportador das variáveis de requisição de uma aplicação web. Através do objeto **request**, podemos acessar três propriedades principais: **request.env** (que armazena variáveis de ambiente, sessão, cache etc), **request.args** e o **request.vars**.

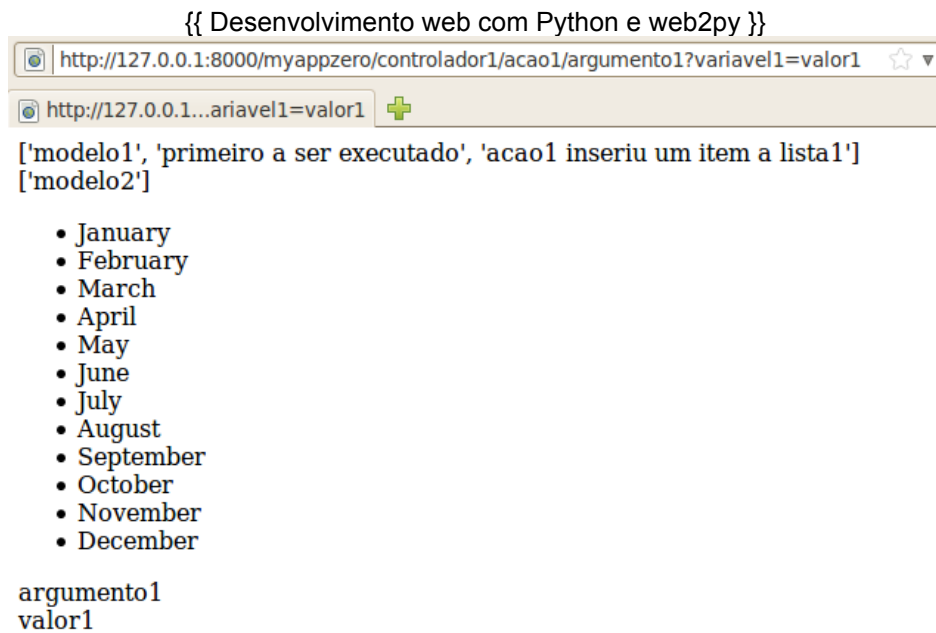
```
1. <html>
2.   <head></head>
3.   <body>
4.     {{print ' '*80}}
5.     {{print "Visão exibida no navegador"}}
6.     {{print request.vars,request.args}}
7.     {{=lista1}}
8.     <br />
9.     {{=lista2}}
10.    <br />
11.    <ul>
12.      {{for mes in meses:
13.        if mes:
14.          }}
15.          <li>{{=mes}}</li>
16.      {{ pass
17.      pass
18.      }}
19.    </ul>
20.    {{print 'acessando argumentos'}}
21.    {{=request.args[0]}}
22.    <br />
23.    {{print 'acessando variáveis'}}
24.    {{=request.vars['variavel1']}}
25.  </body>
26. </html>
```

Acesse o endereço:

<http://127.0.0.1:8000/myappzero/controlador1/acao1/argumento1?variavel1=valor1>

Repare que estamos passando um primeiro argumento com o valor **argumento1** e uma queryString com a chave **variavel1** de **valor1**.

A saída em seu navegador deverá ser:



ciclo de vida da aplicação

E a saída da janela de terminal deverá ser:

```

1. >, 'T': <gluon.languages.translator object at 0xceeee6c>, 'IS_IN_SET': <class
2. 'gluon.validators.IS_IN_SET'>,
3.
4. 'IS_UPLOAD_FILENAME': <class 'gluon.validators.IS_UPLOAD_FILENAME'>,
5. 'IS_DECIMAL_IN_RANGE':
6.
7. <class 'gluon.validators.IS_DECIMAL_IN_RANGE'>}
8.
9. inicio do ciclo de vida da aplicação web2py
10. criando uma lista global
11. ['modelo1', 'primeiro a ser executado']
12.
13. segundo modelo a ser executado pelo web2py
14. criando outra lista global
15. ['modelo2', 'segundo a ser executado']
16. importando o módulo calendário e imprimindo os meses
17. ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October',
18. 'November',
19. 'December']
20.
21. Inicio da execução da acao1 no controlador1
22. ['modelo1', 'primeiro a ser executado', 'acao1 inseriu um item a lista1']
23. acao1 excluiu um item de lista2
24. ['modelo2']
25. acao1 irá se comunicar com a visão
26.
27. Visão exibida no navegador
    <Storage {'variavel1': 'valor1'}> ['argumento1']
    acessando argumentos
    acessando variáveis

```

A camada de acesso a dados - DAL

O web2py possui uma camada de abstração de banco de dados (DAL), que é uma API que mapeia objetos Python em objetos de banco de dados como queries, tabelas e registros. A DAL gera códigos SQL dinamicamente, em tempo real, utilizando sempre o dialeto SQL referente ao banco de dados em uso. Dessa forma, você não precisa escrever código SQL ou aprender comandos SQL de um banco de dados específico, e sua aplicação será portátil para diferentes bancos de dados. Atualmente, os bancos de dados suportados são: SQLite (que já vem com Python e com web2py), PostgreSQL, MySQL, Oracle, MSSQL, FireBird, DB2, Informix e Ingres. O web2py também é capaz de se conectar ao Google BigTable no Google App Engine (GAE).

O framework define as seguintes classes para construir a DAL:

DAL representa a conexão com o banco de dados

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

O código acima define uma referência **db** para uma instância da classe **DAL**, mapeada para um banco de dados do tipo **sqlite** com o nome **storage.db**, por padrão armazenado na pasta **databases**.

Table representa uma tabela no banco de dados, você não pode instanciar Table diretamente: para isso usamos o método DAL.define_table

```
1. db.define_table('minhatabela', Field('meucampo'))
```

O código acima define uma tabela **minhatabela**, contendo apenas um campo **meucampo**. O web2py se encarrega de criar e executar o código SQL responsável pela criação desta tabela. O objeto **db** agora possui uma propriedade chamada **minhatabela** e o acesso é feito com **db.minhatabela**.

Os métodos mais importantes da classe **Table** são: **.insert**, **.truncate**, **.drop**. Ex:

```
1. db.minhatabela.truncate()
```

Field representa um campo da tabela, ele pode ser instanciado ou passado como argumento para o método DAL.define_table

```
1. db.define_table('minhatabela',  
2.                 Field('meucampo', 'string', notnull=True, required=True,\  
3.                 requires=IS_NOT_EMPTY(), default='Olá Mundo'),  
4.                 )
```

No exemplo anterior, **Field** recebe vários argumentos que definem as propriedades deste campo na tabela. Este código pode ser traduzido para o SQL **CREATE TABLE** minhatabela(meucampo varchar(255) not null default 'Olá Mundo'

O web2py possui uma série de validadores. Alguns agem no nível do banco de dados alterando assim a cláusula SQL, como é o caso no validador **notnull=True**, que se transforma em **not null** no SQL. Outros validadores, como o **required=True**, atuam no nível da DAL definindo, por exemplo, que é requerida a passagem de um valor para ser inserido neste campo. Já outros, como o **IS_NOT_EMPTY**, funcionam no nível do formulário, desta forma, quando criamos um formulário, este form será validado garantindo que este campo não seja vazio.

A definição de validadores pode ser feita separadamente da definição da tabela, e esta é uma prática recomendada:

```
1. db.define_table('minhatabela',
2.                 Field('meucampo', 'string'),
3.                 )
4.
5. db.minhatabela.notnull=True
6. db.minhatabela.required=True
7. db.minhatabela.requires=IS_NOT_EMPTY()
8. db.minhatabela.default='Olá Mundo'
```

DAL Rows é o objeto retornado por um select, considere-o como uma lista de registros.

```
1. registros = db(db.minhatabela.meucampo!=None).select()
```

O método **select** executa a cláusula **SELECT** do SQL, utilizando as condições informadas em **db()**. Este mesmo código em ANSI SQL ficaria: ****SELECT * FROM minhatabela where meucampo is not null****

Row contém campos com valores:

```
1. for registro in registros:
2.     print registro.meucampo
```

Query é um objeto que representa a cláusula SQL where

```
1. minhaquery = (db.minhatabela.meucampo == 'Teste')
```

Este código poderia ser traduzido para o fragmento **minhatabela where meu campo = 'Teste'**, e esta query pré-definida poderá ser aplicada ao objeto **Set**.

Set é um objeto que representa um conjunto de registros.

Seus métodos mais importantes são count, select, update e delete. Por exemplo:

```
1. # definir o conjunto de registros 'SELECT * FROM minhatabela where meucampo = 'Teste'
2. conjunto = db(minhaquery)
3. # executar o select acima e popula o objeto registros
4. registros = conjunto.select()
5. # alterar os registros 'UPDATE minhatabela set meucampo = 'Teste2' where meucampo =
6. 'Teste'
7. conjunto.update(meucampo='Teste2')
8. # deletar os registros 'DELETE FROM minhatabela where meucampo = 'Teste'
   conjunto.delete()
```

Expressions são usadas para representar, por exemplo, as expressões orderby ou groupby:

```
1. minhaordem = db.minhatabela.meucampo.upper() | db.minhatabela.id
2. db().select(db.minhatabela.ALL, orderby=minhaordem)
```

Representação de registros

Este argumento é opcional, porém, é muito útil para a criação dos formulários, pois esta representação será utilizada para popular os drop-downs de referência.

Exemplo:

```
1. db.define_table('pessoa',
2.                 Field('nome'),
3.                 format='%(name)s',
4.                 )
5. # ou
6.
7. db.define_table('pessoa',
8.                 Field('nome'),
9.                 format='%(name)s %(id)s',
10.                )
11.
12. # ou um pouco mais complexo
13.
14. db.define_table('pessoa',
15.                 Field('nome'),
16.                 format=lambda r: r.nome or 'anônimo',
17.                 )
18.
```

Nos três casos acima, estamos definindo qual será o padrão de representação desta tabela, quando ela for referenciada em um drop-down. Ao invés de utilizar o campo **id**, o web2py usará a representação definida em **format**.

Interagindo com a DAL

É possível utilizar a DAL através do console; abra um terminal e digite:

```
1. python web2py.py -S welcome
```

- O **-S** indica que o web2py será iniciado no modo **Shell**
- **welcome** é o nome da aplicação que iremos iniciar

Será exibido o console interativo do Python, porém, o ambiente web2py estará carregado na memória e a aplicação **welcome** será instanciada. Como estamos no modo de console, precisamos importar manualmente o que iremos utilizar. Importe todo o conteúdo do pacote **gluon**:

```
1. >>> from gluon import *
```

Vamos criar e conectar com um banco de dados **SQLite**, utilizando a **DAL**:

```
1. >>> banco = DAL('sqlite://meubanco.db')
2. >>> banco.tables
3. []
```

Vamos criar uma tabela chamada **carros**, contendo os campos **marca** e **modelo**:

```
1. >>> banco.define_table('carros',Field('marca'),Field('modelo'))
```

Você verá o código SQL gerado pela DAL.

```
1. <Table {'ALL': <gluon.sql.SQLALL object at 0xa28c50c>, '_sequence_name': None,
2. '_referenced_by': [], 'fields': ['id', 'marca', 'modelo'], '_db':
3. <SQLDB {'_connection': <sqlite3.Connection object at 0xa1d8220>,
4. '_lastsql': 'CREATE TABLE carros(\n  id INTEGER PRIMARY KEY AUTOINCREMENT,\n
5.   marca CHAR(512),\n  modelo CHAR(512)\n
6. ....
7. ....
```

Para listar as tabelas existentes em **banco**, execute:

```
1. >>> banco.tables
2. ['carros']
```


Para listar os campos da tabela **carros**, execute:

1. `>>> banco.carros.fields`
2. `['id', 'marca', 'modelo']`

Insert, Select, Update, Delete

Inserindo registros

Para inserir registros no banco de dados, usamos o método `insert` da classe **Table**:

1. `>>> banco.carros.insert(marca='GM', modelo='Corsa')`
2. `1`
3. `>>> banco.carros.insert(marca='AUDI', modelo='A4')`
4. `2`

Apesar de o método **insert** retornar o **id** do registro que foi criado, este registro está apenas na memória e ainda não foi inserido no banco de dados físico. Para executar a inserção, precisamos explicitamente executar o método **commit** e caso queiramos desfazer esta ação, utilizamos o método **rollback**.

1. `>>> banco.commit() # dados inseridos no banco SQLite`

Você pode inserir vários registros ao mesmo tempo, utilizando o método **bulk_insert**:

1. `>>> novoscarros = [{'modelo': 'Corolla', 'marca': 'Toyota'}, \`
2. `{'modelo': 'Fusca', 'marca': 'Volks'}, \`
3. `{'modelo': 'Clio', 'marca': 'Renault'}]`
4. `>>> banco.carros.bulk_insert(*[`
5. `{'marca': carro['marca'],`
6. `'modelo': carro['modelo']} for carro in novoscarros])`
7. `[3, 4, 5]`

Consultando os registros

Existem duas formas de consultar todos os registros de uma tabela ****SELECT * FROM ****

- **Utilizando `base.tabela.ALL`**

```
1. >>> registros = banco().select(banco.carros.ALL)
2. >>> for registro in registros:
3. ...     print registro.marca, registro.modelo
4. ...
5. GM Corsa
6. AUDI A4
7. Toyota Corolla
8. Volks Fusca
9. Renault Clio
```

O Código SQL gerado pelo comando acima é:

```
1. >>> banco._lastsql
2. 'SELECT carros.id, carros.marca, carros.modelo FROM carros;'
```

DAL._lastsql retorna o último comando SQL executado pela DAL

- **Utilizando uma `query`**

```
1. >>> registros = banco(banco.carros.id>0).select()
```

Deste modo, o operador **id>0** poderia ser substituído, utilizando os operadores (**==**, **!=**, **<**, **>**, **<=**, **>=**, **like**, **belongs**).

Neste caso, o comando SQL seria: **'SELECT carros.id, carros.marca, carros.modelo FROM carros WHERE carros.id>0;'**

Query

Para facilitar, você pode armazenar a referência para uma tabela em uma variável.

```
1. >>> carros = banco.carros
```

Você também pode armazenar a referência para um campo em uma variável:

```
1. >>> marca = carros.marca
```

Você pode também definir uma **query** utilizando os operadores (**==**, **!=**, **<**, **>**, **<=**, **>=**, **like**, **belongs**) e armazenar esta query em uma variável:

```
1. >>> query = marca!= 'Toyota'
```

Quando você invoca o banco de dados com uma query, você define um conjunto de registros, que também pode ser armazenado em uma variável:

```
1. >>> conjunto = banco(query)
```

O comando anterior apenas define o conjunto de dados que será retornado, porém, ainda não executa a query no banco de dados. Para executar esta query, é necessário chamar explicitamente:

```
1. >>> registros = conjunto.select()
2. >>> for registro in registros:
3. ...     print registro.marca, registro.modelo
4. ...
5. GM Corsa
6. AUDI A4
7. Volks Fusca
8. Renault Clio
```

O código SQL que foi executado acima é: **"SELECT carros.id, carros.marca, carros.modelo FROM carros WHERE carros.marca<>'Toyota';"**

Atalhos

A DAL suporta vários tipos de atalho, em particular:

```
1. >>> meuregistro = banco.carros[id]
```

Seria o mesmo que:

```
1. >>> meuregistro = banco(banco.carros.id==1).select().first()
```

Outros atalhos funcionais são:

```
1. >>> registro = banco.carros(1)
2. >>> registro = banco.carros(banco.carros.id==1)
3. >>> registro = banco.carros(1, marca='GM')
```

Ordenação

Você pode consultar os registros ordenados por marca:

```
1. >>> for registro in banco().select(banco.carros.ALL, orderby=banco.carros.marca):
2. ...     print registro.marca,registro.modelo
3. ...
4. AUDI A4
5. GM Corsa
6. Renault Clio
7. Toyota Corolla
8. Volks Fusca
```

Query múltipla

Podemos utilizar mais de um argumento como filtro de uma query. Para isso, podemos utilizar | (ou) e & (e)

```
1. >>> for registro in banco((banco.carros.marca=='Toyota') |
2. (banco.carros.modelo=='Clio')).select():
3. ...     print registro.marca,registro.modelo...
4. Toyota Corolla
   Renault Clio
```

Contagem de registros

Para contar a quantidade de registros em um conjunto, utilizamos o método **count()**:

```
1. >>> banco(banco.carros.id>0).count()
2. 5
```

Alteração de registros

Para alterar um registro existente no banco de dados, podemos usar:

```
1. >>> carro = banco.carros[2]
2. >>> carro.marca
3. 'AUDI'
4. >>> carro.modelo
5. 'A4'
6. >>> carro.modelo = 'A3'
7. >>> carro.modelo
8. 'A3'
```

Ou então, podemos atualizar vários registros de uma única vez:

```
1. >>> carros = banco().select(banco.carros.ALL)
2. >>> for carro in carros:
3. ...     carro.update(marca='FIAT')
```

Podemos também excluir os registros:

```
1. >>> banco(banco.carros.id>0).delete()
2. 5
```

Tipos de dados

A DAL suporta os seguintes tipos de dados:

- Field(name, 'string')
- Field(name, 'text')
- Field(name, 'password')
- Field(name, 'blob')
- Field(name, 'upload')
- Field(name, 'boolean')
- Field(name, 'integer')
- Field(name, 'double')
- Field(name, 'time')
- Field(name, 'date')
- Field(name, 'datetime')
- Field(name, db.referenced_table) # reference field
- Field(name, 'list:string')

Atributos

Cada campo pode receber os seguintes atributos de acordo com o seu tipo de dado:

- length (apenas para o tipo string, padrão 32)
- default (padrão None)
- required (padrão False)
- notnull (padrão False)
- unique (padrão False)
- requires (um validador ou uma lista de validadores para os formulários)
- comment (comentários para formulários)
- widget (para formulários)
- represent (para formulários)
- readable (determina se o campo é visível no formulário, padrão True)
- writable (determina se o campo é alterável no formulário, padrão True)
- update (valor padrão caso o registro seja atualizado)

- `uploadfield` (para criação de um campo BLOB para upload, padrão `None`)
- `authorize` (para autenticar em caso de download protegido)
- `autodelete` (padrão `False`, caso `True`, apaga a imagem assim que sua referência no banco for apagada)
- `label` (rótulos para os formulários)

Migrações

Alterar a lista de campos ou os tipos dos campos em um modelo provoca uma migração automática, ou seja, o web2py gera código SQL automaticamente para alterar a tabela de acordo com as mudanças. Se a tabela não existir, ela é criada. Ações de migração são registradas no arquivo `Sql.log`, acessível através da interface administrativa. A migração pode ser desligada por tabela em uma base de dados, passando o parâmetro **`migrate = False`** no método **`define_table`**.

Validadores

Validadores são classes usadas para validar campos de entrada em formulários (incluindo os gerados através do banco de dados).

Exemplo de uso de um validador na criação de um formulário.

1. `INPUT(_name='marca', requires=IS_NOT_EMPTY(error_message='Você deve informar a marca!'))`

Aqui um exemplo de uso de validador na base de dados:

1. `banco.define_table('carros', Field('marca'))`
2. `banco.carros.marca.requires = IS_NOT_EMPTY()`

Os validadores sempre são atribuídos através do atributo **`requires`** da classe **`Field`**. Um campo pode ter um ou mais validadores. Múltiplos validadores são formados com uma **lista**

1. `bancos.carros.marca.requires = [IS_NOT_EMPTY(),`
2. `IS_NOT_IN_DB(banco, 'carros.marca')]`

Os validadores são invocados através da função **`accepts`** em um **FORM** ou qualquer outro **HTML helper** que contenha um **FORM**. São invocados na exata ordem em que são listados.

Os validadores de formulário

IS_ALPHANUMERIC

Este validador checa se um campo contém um caractere dentro do range **a-z**, **A-Z** ou **0-9**

IS_DATE

Este validador checa se o campo contém uma data válida, de acordo com o formato especificado

- 1.
2. `requires = IS_DATE(format='%Y-%m-%d',`
3. `error_message='O formato deve ser AAAA-MM-DD')`

IS_DATE_IN_RANGE()

Funciona da mesma maneira que o **IS_DATE**, mas permite especificar um range de datas

1. `requires = IS_DATE_IN_RANGE(format='%Y-%m-%d',`
2. `minimum=datetime.date(2008,1,1),`
3. `maximum=datetime.date(2009,12,31),`
4. `error_message='O formato deve ser AAAA-MM-DD'`
5. `)`

IS_IN_SET

Checa se o valor está contido em um conjunto (set):

1. `requires = IS_IN_SET(['a', 'b', 'c'],`
2. `zero='escolha um valor',`
3. `error_message='o valor deve ser a, b ou c'`
4. `)`

IS_LENGTH

Verifica se o valor respeita o tamanho mínimo e máximo especificados.

Os argumentos são:

- **maxsize:** O tamanho máximo permitido
- **minsize:** O tamanho mínimo permitido

Exemplo: checa se o texto é menor do que 10 caracteres:

1. `INPUT(_type='text', _name='marca', requires=IS_LENGTH(9))`

Verificando se a senha é maior do que 5 caracteres:

1. `INPUT(_type='password', _name='senha', requires=IS_LENGTH(minsize=6))`

IS_LOWER

Este validador nunca retorna um erro, ele apenas converte o valor inserido para texto em minúsculo.

1. `requires = IS_LOWER()`

IS_NOT_EMPTY

Verifica se o valor do campo não está vazio.

1. `requires = IS_NOT_EMPTY(error_message='Não pode ser vazio')`

IS_URL

Verifica se o valor respeita a sintaxe de uma URL válida.

[<http://www.web2pybrasil.com.br> , <http://www.temporealeventos.com.br>]

1. requires = `IS_URL()`
2. requires = `IS_URL(mode='generic')`
3. requires = `IS_URL(allowed_schemes=['https'])`
4. requires = `IS_URL(prepend_scheme='https')`
5. requires = `IS_URL(mode='generic',`
6. `allowed_schemes=['ftps', 'https'],`
7. `prepend_scheme='https')`

IS_IMAGE

Em um campo de upload checka se o formato do arquivo é uma imagem válida. Podemos também especificar as dimensões mínima e máxima para o arquivo.

1. requires = `IS_IMAGE(extensions=('jpeg', 'png'), maxsize(200, 200), minsize(50, 50))`

IS_EMPTY_OR

Se você precisar que um campo aceite um valor vazio e ainda assim quiser efetuar outras validações, por exemplo, em um campo que tem que ser uma **data** ou estar vazio:

1. requires = `IS_EMPTY_OR(IS_DATE())`

Os validadores de banco de dados

IS_NOT_IN_DB

Considere o seguinte exemplo:

1. banco.define_table('pessoa', Field('nome'))
2. banco.pessoa.nome.requires = `IS_NOT_IN_DB(banco, 'pessoa.nome')`

Quando você tentar inserir uma nova **pessoa**, será validado se o nome já não existe no banco de dados **banco**, no campo **pessoa.nome**. Como os outros validadores, esta checagem é feita no nível do formulário, portanto, é aconselhável utilizar sempre em conjunto com o validador **unique** que atua no banco de dados.

1. banco.define_table('pessoa', Field('nome', unique=True))
2. banco.pessoa.nome.requires = `IS_NOT_IN_DB(banco, 'pessoa.nome')`

IS_IN_DB

o **IS_IN_DB** funciona da mesma forma que o anterior, porém checka se o valor existe na tabela.

notnull

O validador **notnull** atua na camada do banco de dados, e é traduzido para o código SQL **NOT NULL**, não permitindo que o campo referido seja de valor nulo.

1. banco.define_table('pessoa', Field('nome', notnull=True))
2. banco.pessoa.nome.requires = `IS_NOT_IN_DB(banco, 'pessoa.nome')`

length

length define o tamanho de um campo do tipo texto:

1. banco.define_table('pessoa', Field('nome', length=55))
2. banco.pessoa.nome.requires = `IS_NOT_IN_DB(banco, 'pessoa.nome')`

default

default define o valor padrão para o campo quando um valor não for definido:

1. banco.define_table('pessoa', Field('nome', default='anônimo'))
2. banco.pessoa.nome.requires = IS_NOT_IN_DB(banco, 'pessoa.nome')

widgets

Auto complete widget

Existem duas maneiras para utilizar o widget **autocomplete**, para autocompletar um campo que recebe uma lista para autocompletar ou a referência de um campo contendo estes valores, ou para autocompletar um campo referenciado.

O primeiro caso é fácil:

Defina no modelo

1. db.define_table('marcas', Field('marca'))
- 2.
3. db.define_table('carros',
4. Field('marca'),
5. Field('modelo'),
6.)
- 7.
8. db.carros.marca.widget = SQLFORM.widgets.autocomplete(
9. request, db.marcas.marca, limitby=(0,5), min_length=2)
- 10.

limitby instrui o widget a mostrar não mais do que cinco sugestões por vez, e **minlength** define que deve ser digitado um número mínimo de caracteres para que a consulta **Ajax** seja executada.

E no controller:

1. **def** index():
2. form=SQLFORM(db.carros)
3. **return** dict(form=form)

Resultado:

form :

Modelo:	<input type="text"/>
Marca:	<div>Vo</div> <div>Volks</div>

Outra forma é definindo a gravação do id ao invés do texto:

```
1. db.define_table('marcas', Field('marca'))
2.
3. db.define_table('carros',
4.                 Field('marca'),
5.                 Field('modelo'),
6.                 )
7.
8. db.carros.marca.widget = SQLFORM.widgets.autocomplete(
9.     request, db.marcas.marca, id_field=db.marcas.id)
10.
```

Appadmin

Para cada aplicação criada, o web2py provê um controlador chamado **appadmin**. Este controlador tem como objetivo auxiliar na administração dos bancos de dados da aplicação.

O controlador **appadmin** é acessível através da URL:

<http://127.0.0.1:8000/<suaapp>/appadmin/index>

Através deste controlador é possível:

- Visualizar a lista de tabelas definidas
- Executar queries da DAL
- Executar operações CRUD nos registros do banco de dados (Inserir, Selecionar, Atualizar, Apagar)
- Exportar o conteúdo de uma tabela
- Importar conteúdo para uma tabela

Lista de bancos de dados e tabelas definidas

Bancos de dados e tabelas disponíveis

db.auth_user

[inserir novoauth_user]

db.auth_group

[inserir novoauth_group]

db.auth_membership

[inserir novoauth_membership]

db.auth_permission

[inserir novoauth_permission]

db.auth_event

[inserir novoauth_event]

db.marcas

[inserir novomarcas]

Executando queries DAL

Linhas na tabela

Consulta:	<input type="text" value="db.marcas.marca=='Toyota'"/>
Atualizar:	<input type="checkbox"/> <input type="text"/>
Apagar:	<input type="checkbox"/>
<input type="button" value="submit"/>	

Uma "consulta" é uma condição como "db.tabela1.campo1=='valor'". Expressões como "db.tabela1.campo1==db.tabela2.campo2" resultam em um JOIN SQL. Use (...)&(…) para AND, (...)|(…) para OR, e ~(…) para NOT para construir consultas mais complexas. "update" é uma expressão opcional como "campo1='novovalor'". Você não pode atualizar ou apagar os resultados de um JOIN

1 selecionado

marcas.id	marcas.marca
1	Toyota

Inserindo um novo registro

banco de dados db tabela **marcas**

Novo Registro

Marca:	<input type="text" value="Toyota"/>
<div>value already in database</div>	
<input type="button" value="Submit"/>	

Importação e exportação de registros

Importar/Exportar

[exportar como um arquivo csv]

ou importar de um arquivo csv

Log SQL

A aplicação **admin** armazena o log de todo o código SQL executado pela DAL.
<http://127.0.0.1:8000/admin/default/peek/<suaapp>/databases/sql.log>

```
1. timestamp: 2010-08-14T18:12:01.902993
2. CREATE TABLE auth_event(
3.     id INTEGER PRIMARY KEY AUTOINCREMENT,
4.     time_stamp TIMESTAMP,
5.     client_ip CHAR(512),
6.     user_id INTEGER REFERENCES auth_user(id) ON DELETE CASCADE,
7.     origin CHAR(512),
8.     description TEXT
9. );
10. success!
11. timestamp: 2010-08-14T18:12:01.913395
12. CREATE TABLE marcas(
13.     id INTEGER PRIMARY KEY AUTOINCREMENT,
14.     marca CHAR(512) NOT NULL UNIQUE
15. );
16. success!
```

Mão na Massa - Visão Geral

O Projeto

Nosso cliente é a **lojadecarro.com** e precisamos desenvolver um novo site para ele. Este site tem como objetivo exibir uma vitrine dos carros disponíveis na loja e possibilitar que o cliente entre em contato demonstrando interesse de compra por algum veículo. Nosso cliente deseja uma interface para incluir, alterar e remover veículos. E quer também ter a possibilidade de inserir uma foto para cada veículo.

Os veículos serão divididos nas categorias **Novos** e **Usados**, e temos a preocupação de tentar utilizar URLs simplificadas para otimizar as buscas.

Por falar em busca, nosso cliente viu um site com uma busca que **completa** o texto do resultado conforme a digitação, e agora ele deseja uma busca igual para o seu site (com **Ajax**).

Loja de Carro

Requisitos

Vendedor

- Incluir um novo veículo
- Alterar um veículo existente
- Remover um veículo
- Consultar compradores
- Consultar mensagens enviadas pelo site

Comprador

- Pesquisar um veículo
- Visualizar detalhes do veículo
- Informar interesse de compra por um veículo
- Entrar em contato com a loja

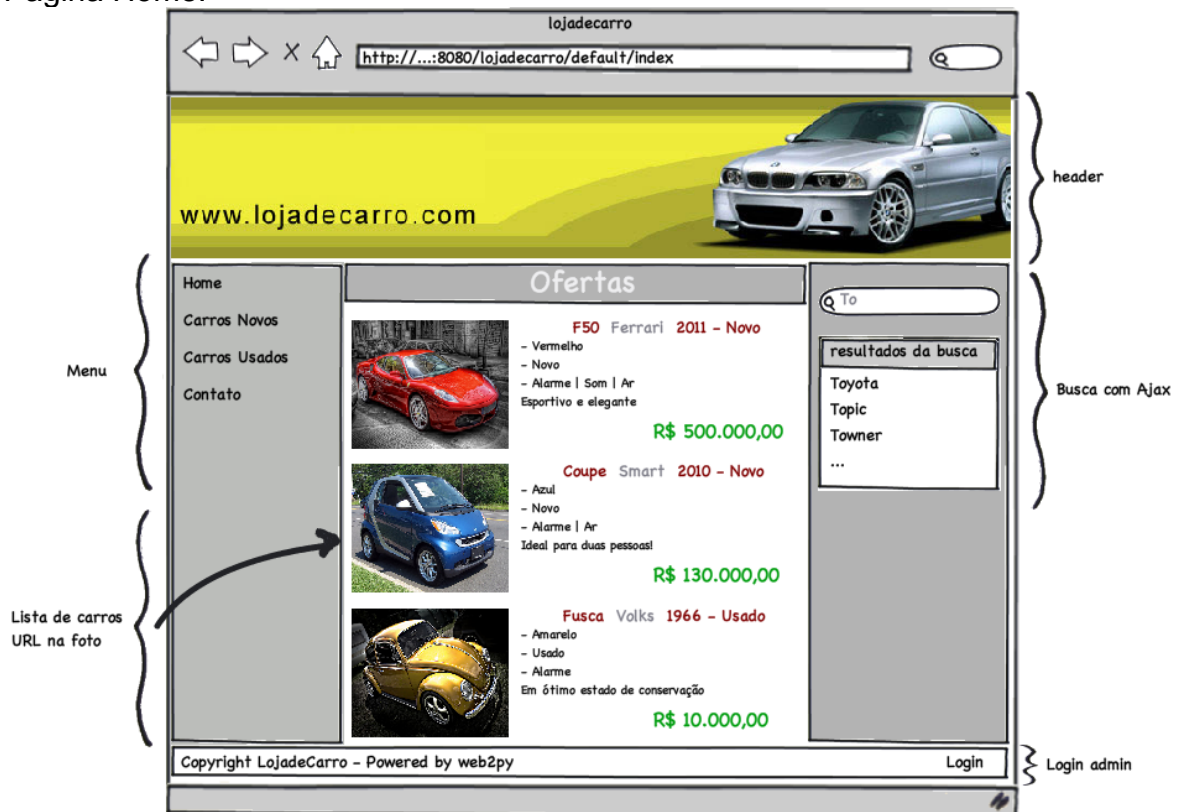
A aplicação

Na interface administrativa do **web2py** crie uma nova aplicação chamada **lojadecarro**. Vamos utilizar a aplicação de modelo do web2py como base para este desenvolvimento.

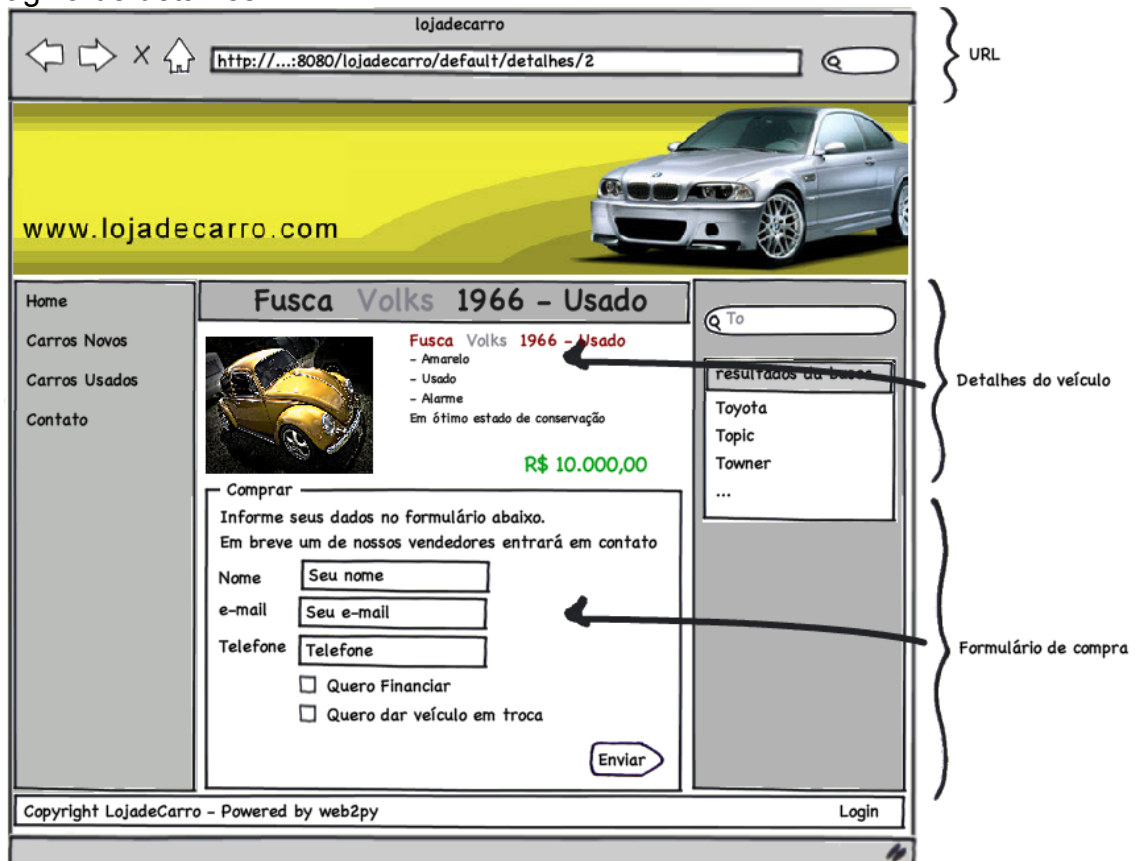
Layout

O cliente também forneceu uma idéia de como ele deseja que seja o layout da aplicação, desta forma, ficará mais fácil entender os requisitos.

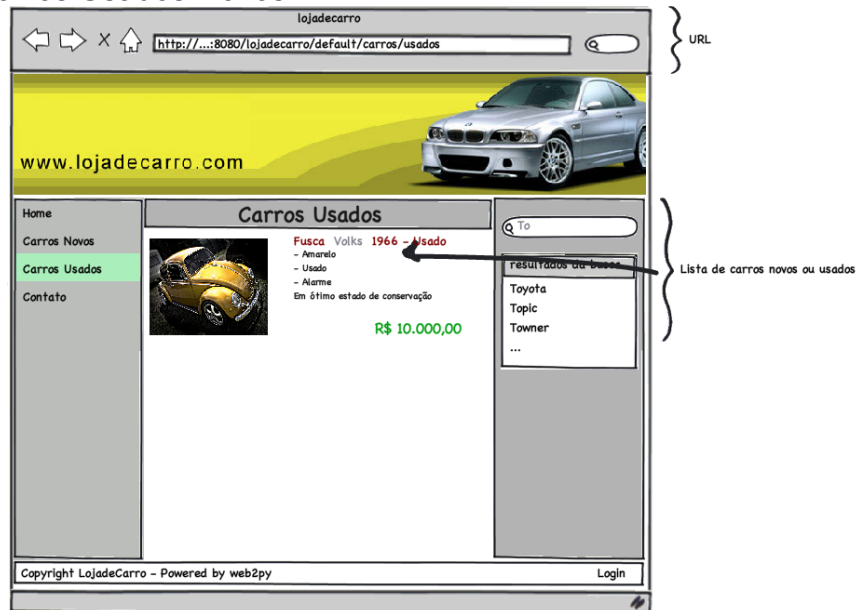
Página Home:



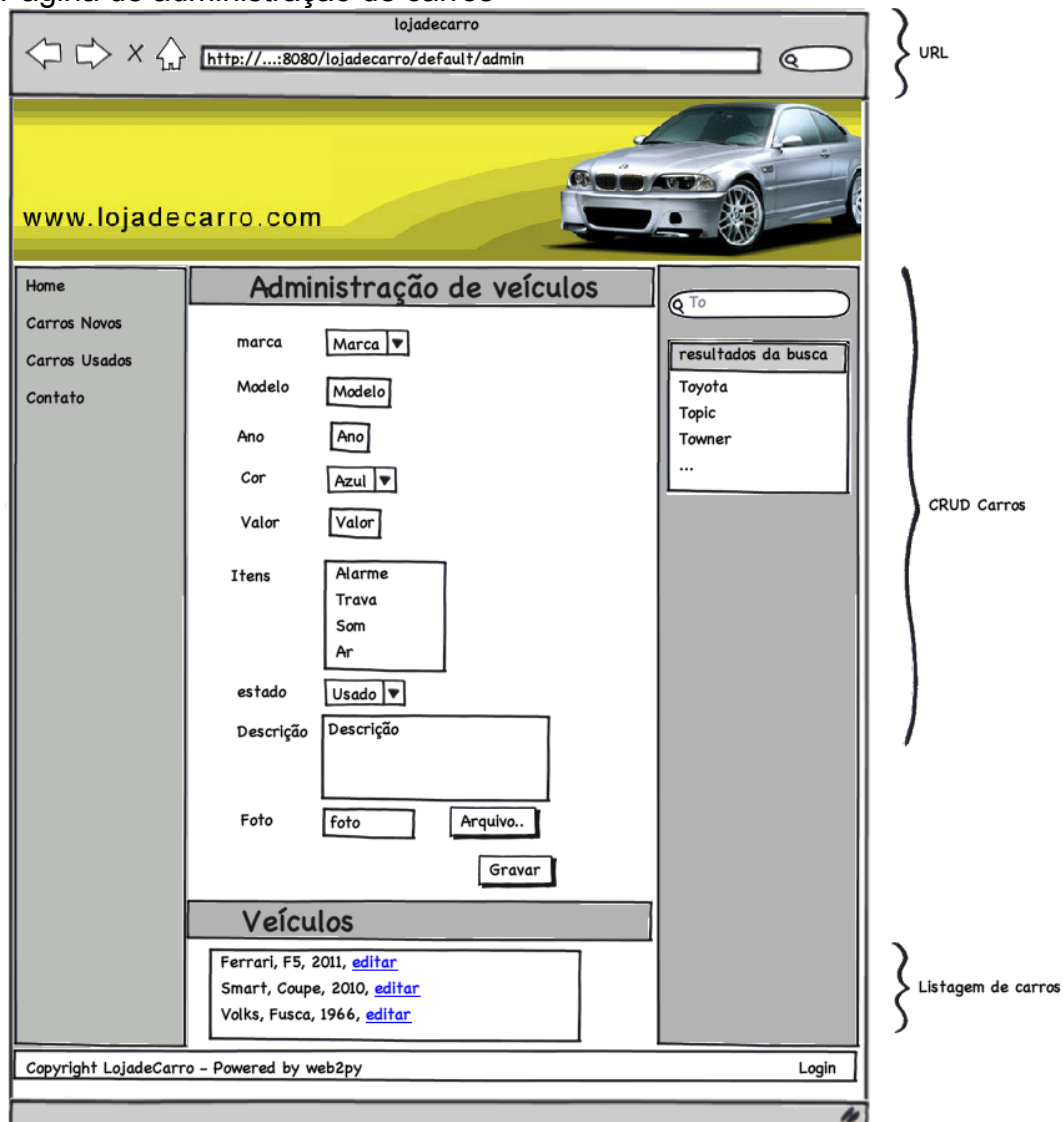
Página de detalhes



Página de carros Usados/Novos



Página de administração de carros



Modelagem do banco de dados

Com base na especificação e no layout, vamos começar pela modelagem das tabelas do nosso banco de dados:

Primeiro, na interface administrativa da aplicação **lojadecarro**, delete o modelo existente **db.py**. Vamos fazer desde o início.

Lembre-se que o **web2py** executa os modelos em ordem alfabética, por este motivo sempre que temos código que precisa ser executado no início do ciclo de vida da aplicação, é recomendada a criação de arquivos de modelo nomeados com números de 0 a 10.

Crie um arquivo de modelo chamado **0**, o web2py automaticamente criará um arquivo **0.py**. Como este arquivo será o primeiro a ser executado pela aplicação, vamos definir o banco de dados e algumas configurações globais.

```
1. arquivo 0.py
2. # coding: utf8
3. db = DAL('sqlite://storage.sqlite')
4. e_m = {
5.     'empty': 'Este campo é obrigatório',
6.     'in_db': 'Este registro já existe no banco de dados',
7.     'not_in_db': 'Este registro não existe no banco de dados',
8.     'email': 'Você precisa inserir um e-mail válido',
9.     'image': 'O arquivo precisa ser uma imagem válida',
10.    'not_in_set': 'Você precisa escolher um valor válido',
11.    'not_in_range': 'Digite um número entre %(min)s e %(max)s',
12. }
13.
14. config = dict(nmsite='Loja de Carro', dscsite='Só os melhores carros')
15.
16. estados = ('Novo', 'Usado')
17.
18. cores = ('Azul', 'Amarelo', 'Verde', 'Vermelho', 'Prata', 'Branco', 'Preto', 'Vinho')
19.
```

Em **0.py** criamos a conexão com o banco de dados através da classe **DAL** e referenciamos por um objeto que chamamos de **db**.

Criamos um dicionário **e_message**, contendo mensagens de validação que usaremos nos formulários da aplicação, e um dicionário **config**, contendo o **nome** e a **descrição** da loja.

Também definimos dois **sets** **estados** e **cores** para utilizar na validação dos formulários.

Podemos utilizar o dicionário **config** para armazenar qualquer tipo de configuração global da aplicação.

Agora crie um novo arquivo de modelo chamado **carros**, o web2py irá criar um arquivo **carros.py** e redirecionará para a tela de edição do modelo.

Neste modelo vamos definir todas as tabelas e validações necessárias para a aplicação. Na hora de definir objetos e tabelas no arquivo de modelo, devemos sempre lembrar que Python é uma linguagem interpretativa, ou seja, os comandos são executados respeitando-se a ordem em que foram definidos no código.

```
1. arquivo carros.py
2. # coding: utf8
3. # criamos um validador pré definido
4. notempty=IS_NOT_EMPTY(error_message=e_m['empty'])
5.
6. # definição da tabela de marcas
7. db.define_table('marca',
8.                 Field('nome', unique=True, notnull=True),
9.                 format='%(nome)s')
10.
11. # validadores da tabela de marcas
12. db.marca.nome.requires=[notempty, IS_NOT_IN_DB(db, 'marca.nome',
13.                                                error_message=e_m['in_db'])]
14.
15.
16. # definição da tabela de carros
17. db.define_table('carro',
18.                 Field('marca', db.marca, notnull=True),
19.                 Field('modelo', notnull=True),
20.                 Field('ano', 'integer', notnull=True),
21.                 Field('cor', notnull=True),
22.                 Field('valor', 'double'),
23.                 Field('itens', 'list:string'),
24.                 Field('estado', notnull=True),
25.                 Field('desc', 'text'),
26.                 Field('foto', 'upload'),
27.                 format='%(modelo)s - %(ano)s - %(estado)s'
28.                 )
29.
30. # validação da tabela carro
31. db.carro.marca.requires=IS_IN_DB(db, 'marca.id', 'marca.nome',
32.                                  error_message=e_m['not_in_db'])
33. db.carro.modelo.requires=notempty
34. db.carro.ano.requires=[notempty, IS_INT_IN_RANGE(request.now.year-
35.                                                    20, request.now.year+2,
36.                                                    error_message=e_m['not_in_range'])]
37. db.carro.cor.requires=IS_IN_SET(cores)
38. db.carro.itens.requires=IS_IN_SET(('Alarme', 'Trava', 'Som', 'Ar'), multiple=True,
39.                                   error_message=e_m['not_in_set'])
40. db.carro.estado.requires=IS_IN_SET(estados, error_message=e_m['not_in_set'])
41. db.carro.foto.requires=IS_EMPTY_OR(IS_IMAGE(extensions=('jpeg', 'png', '.gif'),
42.                                              error_message=e_m['image']))
43.
44.
45.
46.
```

```
47. # definição da tabela de compradores
48. db.define_table('comprador',
49.                 Field('id_carro', db.carro),
50.                 Field('nome'),
51.                 Field('email'),
52.                 Field('telefone'),
53.                 Field('financiar', 'boolean'),
54.                 Field('troca', 'boolean'),
55.                 Field('data', 'datetime', default=request.now)
56.                 )
    # validação da tabela de compradores
db.comprador.nome.requires=notempty
db.comprador.email.requires=IS_EMAIL(error_message=e_m['email'])
db.comprador.telefone.requires=notempty
```

Neste ponto já será possível, através do **appadmin**, inserir alguns registros no banco de dados. Insira algumas marcas como: Toyota, FIAT, GM, Volks, Ferrari, Smart.



Insira também cerca de dois registros de veículos na tabela de carros.

banco de dados db tabela carro

Novo Registro

Marca:	<input type="text" value="GM"/>
Modelo:	<input type="text" value="Agile"/>
Ano:	<input type="text" value="2011"/>
Cor:	<input type="text" value="Azul"/>
Valor:	<input type="text" value="39000"/>
Itens:	<div><div>Alarme</div><div>Trava</div><div>Som</div><div>Ar</div></div>
Estado:	<input type="text" value="Novo"/>
Desc:	<div>Novo lançamento da GM</div>
Foto:	<div><input type="text"/> Enviar arquivo...</div>
<div>Submit</div>	

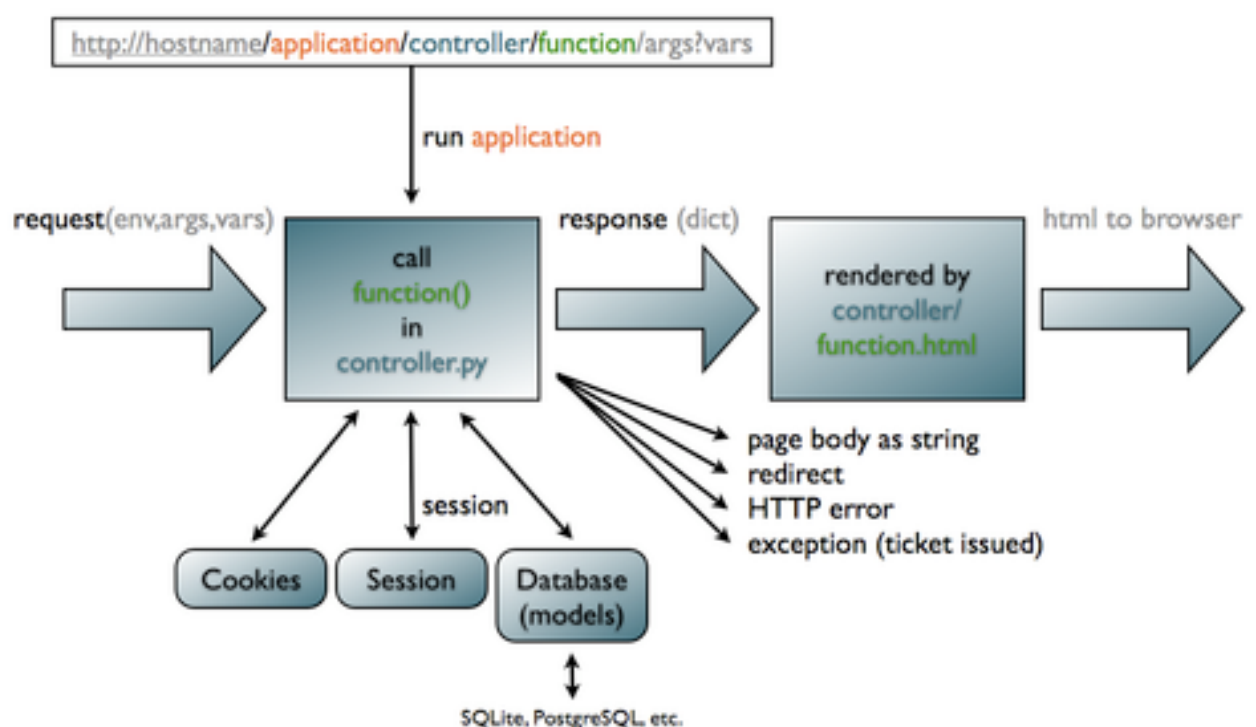
Exercicio:

Analisando a definição de layout, os requisitos e o modelo de dados. Forme uma dupla e desenvolva o restante da aplicação **Loja de Carro**. Com a ajuda do instrutor crie os **controllers** e as **views** para cada uma das páginas apresentadas no layout.

Não se preocupe com design, layout e as barras laterais da aplicação, desenvolva apenas o conteúdo principal.

Mapeamento de URLs

Controladores são a porta de entrada de uma aplicação web2py. Quando um usuário efetua uma requisição (request) no navegador, o web2py irá mapear esta requisição de acordo com o esquema de mapeamento de URLs definido no framework. Este esquema de mapeamento já vem por padrão definido da seguinte maneira:



Antes de começar o ciclo de vida da aplicação, ou seja, no passo zero, o web2py irá procurar a definição deste esquema e, a partir daí, procurar os respectivos itens da URL, respectivamente: *aplicação*, *controlador*, *ação*. A partir desta definição, o web2py irá passar os argumentos e parâmetros para o controlador, que decidirá qual visão irá invocar para exibir o retorno da requisição (response).

Vamos pegar, por exemplo, a seguinte URL: <http://127.0.0.1:8080/a/c/f.html>. O web2py irá procurar um arquivo de configuração de mapeamento de URLs (`routes.py`), e se este arquivo for encontrado, será aplicado o modelo definido nele. Caso este arquivo não exista, então o web2py irá utilizar o mapeamento padrão que segue o seguinte modelo:

Lendo do final para o início:

A função **f()** definida no controlador **c.py** que faz parte da aplicação **a**, e que retorna conteúdo no formato **HTML**.

Caso a função **f** não exista, o web2py redirecionará para a função **index**. Caso **c.py** não exista, o web2py redirecionará para o controlador **default** e caso **a** não exista, o web2py redirecionará para a aplicação **init** e **welcome** respectivamente. Se nenhuma dessas condições forem satisfeitas, então a mensagem **invalid request** será exibida.

Por padrão, a cada nova requisição também serão criadas variáveis de sessão. Adicionalmente, um cookie referente a esta sessão será enviado ao browser do cliente para manter referência a esta sessão.

A extensão **html** é opcional e, mesmo que você não informe, o web2py irá assumir que você quer o retorno no formato html. A extensão em uma requisição determina a extensão da view correspondente que irá renderizar o retorno da função **f()**. E este mesmo conteúdo poderia ser servido em múltiplos formatos (html, xml, json, rss etc.).

Funções que recebem argumentos ou que iniciam com duplo underline **__** não são expostas publicamente e podem ser acessadas apenas através de outras funções. Esta pode ser uma boa prática de acoplamento (boxing).

Existe uma exceção a este esquema de

URLs: <http://127.0.0.1/a/static/nomedeuumarquivo>

Não existe um controlador com o nome **static**, porém o web2py irá mapear esta requisição diretamente para a pasta de arquivos estáticos de sua aplicação. E como estes arquivos não exigem processamento, quando um conteúdo estático é diretamente requisitado, o web2py não criará variáveis de sessão, não enviará cookies ao navegador e não executará os arquivos de modelo. O web2py serve arquivos estáticos em blocos de 1MB, e envia o conteúdo parcialmente ao navegador quando este limite é ultrapassado.

O web2py também implementa o protocolo **IF_MODIFIED_SINCE** (se alterado desde...) e não serve um arquivo estático, caso ele já esteja no *cache* do navegador e não tenha sido alterado desde a última versão.

Passagem de parâmetros

O web2py mapeia os parâmetros POST e GET da seguinte maneira:

<http://127.0.0.1:8000/a/c/f.html/x/y/z?p=1&q=2>

Para a função **f** do controlador **c.py** na aplicação **a**, os parâmetros de URL são armazenados no objeto **request** seguindo a seguinte estrutura:

```
1. request.args = ['x', 'y', 'z']
2.
3. request.vars = {'p':1, 'q':2}
4.
5. request.application = 'a'
6. request.controller = 'c'
7. request.function = 'f'
```

Nos exemplos acima, tanto **request.args[i]** como **request.args(i)** podem ser utilizados para acessar o elemento do objeto request.args, porém, enquanto o primeiro estoura uma exceção caso a lista não possua o índice especificado, o último retorna **None** neste caso.

```
1. request.url
```

request.url armazena a URL completa para a respectiva requisição, não incluindo os valores de POST e GET.

Se a requisição HTTP é GET, então **request.env.request_method** terá o valor "GET"; se for POST, **request.env.request_method** terá o valor "POST". Variáveis **querystring** são armazenadas no dicionário **request.vars** e também são referenciadas em **request.get_vars** (GET) ou **request.post_vars** (POST).

O web2py também armazena variáveis de ambiente WSGI, como:

```
1. request.env.path_info
2. 'a/c/f'
```

E também os cabeçalhos HTTP:

```
1. request.env.http_host
2. '127.0.0.1:8000'
3.
```

O web2py valida toda URL para prevenir ataques. As URLs podem conter apenas caracteres alfanuméricos, underlines, barras, traços; os argumentos não podem conter pontos consecutivos. Todo espaço em branco é substituído por um underline.

Se a URL corresponder a um arquivo estático, o web2py retornará este arquivo, caso contrário processará na seguinte ordem:

- Checagem de cookies
- Criação do ambiente em que será executada a função requisitada
- Inicialização dos objetos request, response, cache
- Abre uma sessão existente ou inicia uma nova sessão
- Executa em ordem alfabética todos os arquivos de modelo da aplicação
- Executa a função requisitada

{{ Desenvolvimento web com Python e web2py }}

- Se a função requisitada retornar um dicionário, executa a respectiva visão
- Em caso de sucesso, efetua a confirmação (commit) de todas as transações abertas
- Salva a sessão
- Retorna um objeto HTTP response

Visões e controladores são executados em diferentes cópias do mesmo ambiente, desta forma, as visões não visualizam os controladores, mas podem ver os modelos e também podem ver as variáveis retornadas pelo controlador.

Se algum erro ou exceção que não seja HTTP acontecer:

- Armazena o retorno do erro e referencia um ticket para este erro
- Reverte (rollback) todas as transações abertas
- Retorna uma tela com o número e link para o ticket de erro

API

Modelos, Visões e Controladores são executados em um ambiente onde são importados e ficam disponíveis os seguintes objetos:

Objetos globais:

request, response, session, cache

Navegação:

redirect, HTTP

internacionalização (i18n):

T

Helpers (bibliotecas auxiliares):

XML, URL, BEAUTIFY

A, B, BEAUTIFY, BODY, BR, CENTER, CODE, DIV, EM, EMBED, FIELDSET, FORM, H1, H2, H3, H4, H5, H6, HEAD, HR, HTML, I, IFRAME, IMG, INPUT, LABEL, LEGEND, LI, LINK, OL, UL, MARKMIN, MENU, META, OBJECT, ON, OPTION, P, PRE, SCRIPT, OPTGROUP, SELECT, SPAN, STYLE, TABLE, TAG, TD, TEXTAREA, TH, THEAD, TBODY, TFOOT, TITLE, TR, TT, URL, XHTML, xmlescape, embed64

Validadores:

CLEANUP, CRYPT, IS_ALPHANUMERIC, IS_DATE_IN_RANGE, IS_DATE, IS_DATETIME_IN_RANGE, IS_DATETIME, IS_DECIMAL_IN_RANGE, IS_EMAIL, IS_EMPTY_OR, IS_EXPR, IS_FLOAT_IN_RANGE, IS_IMAGE, IS_IN_DB, IS_IN_SET, IS_INT_IN_RANGE, IS_IPV4, IS_LENGTH, IS_LIST_OF, IS_LOWER, IS_MATCH, IS_EQUAL_TO,

IS_NOT_EMPTY, IS_NOT_IN_DB, IS_NULL_OR, IS_SLUG, IS_STRONG, IS_TIME, IS_UPLOAD_FILENAME, IS_UPPER, IS_URL

Banco de dados:

DAL, Field

Outros objetos e módulos são definidos nas bibliotecas, mas eles não são automaticamente importados até que sejam utilizados.

As principais entidades da API no ambiente de execução web2py são: **request, response, session, cache, URL, HTTP, redirect** e **T**.

Alguns objetos e funções, incluindo **Auth** (autenticação), **Crud** (formulários) e **Service** (webservices), são definidos no módulo **gluon/tools.py** e necessitam serem importados caso sejam necessários.

1. **from** gluon.tools **import** Auth, Crud, Service

Renderização de conteúdo

Como já foi discutido anteriormente, o web2py mapeia as funções Python (as ações) definidas nos controladores como sendo a porta de entrada para a requisição. Caso esta função retorne um dicionário de dados, o web2py então renderiza a visão correspondente com a função+extensão.

Caso o controlador retorne um objeto string, por exemplo, isto também será renderizado em uma view, pois strings podem ser serializadas como dicionário, e este exemplo serve para qualquer outro objeto com esta característica: é um dicionário ou pode ser serializado como um.

Podemos criar visões específicas para cada ação, podemos utilizar uma única visão para renderizar conteúdo de várias ações e também podemos ter varias visões que renderizam conteúdo de uma única ação.

Caso o web2py não encontre uma visão específica para renderizar o conteúdo de uma ação, então ele renderizará utilizando as visões genéricas; por padrão temos uma view genérica para cada um dos seguintes tipos de conteúdo:

- **/views/generic.html** renderiza conteúdo de ações requisitadas que não possuam extensão ou que tenham a extensão **.html**
- **/views/generic.json** requisições com a extensão **.json**
- **/views/generic.xml** requisições com a extensão **.xml**
- **/views/generic.load** requisições com a extensão **.load** (usado para componentes e Ajax)

Também é possível renderizar o conteúdo em formato **rss** em **/views/generic.rss**, mas este é um caso especial, onde o dicionário retornado pela ação deve respeitar um formato **rss** válido.

Outra possibilidade é a criação de sua própria visão genérica. Implementando seu próprio serializador, ou utilizando uma biblioteca específica, é possível ter uma visão genérica **/views/generic.xls** para renderizar o conteúdo do dicionário com o formato .xls(Microsoft Excel), ou qualquer outro tipo de retorno.

Exemplos:

Em uma aplicação de teste, dentro do controlador **default.py**, defina a seguinte ação:

```
1. def aluno():
2.     agora = request.now
3.     return dict(nome='Fulano', curso='web2py', idade=27, dia=agora.date(),
        hora=agora.time())
```

Agora faça um request em outra aba ou janela do seu navegador para o endereço da função (ação) **aluno**

- <http://127.0.0.1:8080/<suaapp>/default/aluno>

Mesmo não informando a extensão, o web2py assume o padrão e renderiza o conteúdo em formato HTML. Experimente agora:

- <http://127.0.0.1:8080/<suaapp>/default/aluno.html>
- <http://127.0.0.1:8080/<suaapp>/default/aluno.xml>
- <http://127.0.0.1:8080/<suaapp>/default/aluno.load>
- <http://127.0.0.1:8080/<suaapp>/default/aluno.json>

Como RSS contém um formato fixo, o retorno de um controlador em formato RSS precisa obedecer à seguinte estrutura:

```
1. def feed():
2.     return dict(title="meu rss",
3.                 link="http://feed.minhaapp.com",
4.                 description="meu primeiro feed com web2py",
5.                 entries=[
6.                     dict(title="web2py framework",
7.                         link="http://feed.minhaapp.com/web2py",
8.                         description="meu primeiro item - Python é Show!")
9.                 ])
```

Agora acesse:

- <http://127.0.0.1:8080/<suaapp>/default/feed.html>
- <http://127.0.0.1:8080/<suaapp>/default/feed.xml>
- <http://127.0.0.1:8080/<suaapp>/default/feed.json>
- <http://127.0.0.1:8080/<suaapp>/default/feed.load>
- <http://127.0.0.1:8080/<suaapp>/default/feed.rss>

Além das formas explanadas aqui, também é possível renderizar conteúdo através de serviços (web-services), que podem ser SOA, SOAP, XML-RPC, REST, etc., porém, este assunto será abordado no módulo avançado deste treinamento.

response

Outra maneira de enviar conteúdo para a visão é utilizando a função **write** do objeto **response**, que tem por objetivo escrever direta e intrusivamente no corpo do html retornado.

Altere a ação **aluno**

1. **def** aluno():
2. **response.write**('escrito diretamente pelo response')
3. **agora** = **request.now**
4. **return** dict(nome='SeuNome', curso='web2py', idade=27, dia=agora.date(), hora=agora.time())

acesse:

- <http://127.0.0.1:8080/<suaapp>/default/aluno.load>

e repare que o **response.write** escreveu conteúdo no corpo da visão, e que este conteúdo é escrito antes de ser efetuado o retorno do dicionário.

Métodos e propriedades do objeto response

response.write pode receber qualquer objeto **string** ou que possa ser serializado como string.

O objeto **response** é um container transportador de **respostas**; o web2py implementa em seu **layout** padrão o objeto **response** como container para algumas informações importantes que são renderizadas no <HEAD> do HTML:

response.cookies: responsável pelo envio de cookies.

response.download(request, db): método implementado para a criação da funcionalidade de download de arquivos.

response.files: uma lista de arquivos CSS e JS que serão incluídos no header do layout padrão. Para incluir um arquivo CSS ou JS apenas inclua um novo objeto a esta lista. Ele resolverá o problema de duplicidade. A ordem é importante.

response.flash: objeto opcional para armazenar uma mensagem de status a ser exibida nas visões.

response.headers: dicionário para cabeçalhos HTTP.

response.menu: parâmetro opcional para ser usado pelo Helper MENU para a criação de menus.

response.meta: um dicionário contendo as informações de META, geralmente utilizado em otimização para busca (SEO), como por exemplo `response.meta.author`, `response.meta.description`, e/ou `response.meta.keywords`. O conteúdo de META é automaticamente incluído no local da tag META do html, e é o arquivo "web2py_ajax.html" que é responsável por esta inclusão.

response.render(view, vars): um método utilizado para invocar diretamente uma visão através de um controlador.

response.stream(file, chunk_size): quando um controlador retorna este tipo de objeto, o web2py efetua o *stream* deste arquivo em blocos de 1MB.

response.title: opcional. Deve conter o título da página a ser renderizado pela tag HTML <title> no *header* de uma visão.

response._vars: é acessível apenas pela visão, e contém o retorno do controlador para a visão.

response.view: o nome da visão a ser renderizada por padrão naquele controlador.

response.write(text): escreve textos no corpo <body> da visão renderizada. Não existem restrições quanto ao uso de **response** para a passagem de parâmetros. Você pode criar novos itens no dicionário response, mas é recomendado manter apenas os seguintes:

1. `response.title`
2. `response.subtitle`
3. `response.flash`
4. `response.menu`
5. `response.meta.author`
6. `response.meta.description`
7. `response.meta.keywords`
8. `response.meta.*`

Session

Session é um dicionário que armazena valores que ficam disponíveis em todo o escopo da aplicação; como a sessão é iniciada antes da execução dos modelos, o acesso a este objeto é global.

A sintaxe é simples:

1. *#definição*
2. `session.minhavariavel = 'valor'`
- 3.
4. *#acesso*
5. `a = session.minhavariavel`
6. ou
7. `a = session['minhavariavel']`

Redirecionamento

O redirecionamento de url é feito através da função **redirect**, que pode ser chamada em qualquer modelo, controlador ou visão.

```
1. redirect('http://www.web2pybrasil.com.br')
```

O código acima faz com que o web2py invoque uma exceção HTTP que efetua o output deste redirecionamento para o navegador, quebrando o fluxo normal do código no momento em que é chamada.

URL

Uma das funções mais importantes no web2py é a função URL. Ela gera uma URL que pode ser mapeada dinamicamente para ações dos controladores e arquivos estáticos.

Por exemplo:

```
1. URL('aluno')
```

Será mapeada para:

```
1. /<aplicação>/<controlador>/aluno
```

Note que a saída da função URL neste caso depende dinamicamente do nome da aplicação que está sendo executada, do controller que foi invocado e dos parâmetros que tenham sido recebidos.

Por exemplo:

```
1. URL('aluno', args=['x','y'], vars=dict(z='t'))
```

Será mapeado em:

```
1. /<aplicação>/<controlador>/aluno/x/y?z=t
```

É possível especificar explicitamente os parâmetros da função URL, e esta é uma prática recomendada:

1. a = aplicação
2. c = controlador
3. f = função

Desta forma:

```
1. URL(a='minhaapp', c='default', f='aluno')
```

Será:

```
1. /minhaapp/default/aluno
```

[<http://www.web2pybrasil.com.br> , <http://www.temporealeventos.com.br>]

Para os arquivos estáticos armazenados na pasta **static**, o web2py provê um controlador virtual que deve ser usado para gerar URLs para estes arquivos.

1. `URL('static', 'imagem.png')`

Será mapeado diretamente para a URL do arquivo: `/<aplicação>/static/imagem.png`

Autenticação e Controle de Acesso

O web2py inclui um poderoso e customizável sistema de controle de acesso baseado em papéis (RBAC).

Este sistema é implementado na classe **Auth** que necessita e implementa as seguintes tabelas:

- **auth_user** armazena o nome do usuário, o endereço de e-mail, a senha e a situação do registro do usuário (pendente, aceito ou bloqueado)
- **auth_group** armazena os grupos ou os papéis para os usuários. Por padrão cada novo usuário criado possui seu próprio grupo, porém, cada usuário pode fazer parte de múltiplos grupos, e cada grupo pode conter diversos usuários.
- **auth_membership** efetua o relacionamento entre usuários e grupos em uma estrutura **muitos-para-muitos**.
- **auth_permission** efetua o relacionamento entre grupos e permissões. Uma permissão é identificada por um nome e opcionalmente por uma tabela e um registro. Por exemplo, membros de um certo grupo podem executar *update* em um registro específico de uma tabela.
- **auth_event** armazena o log do sistema de autenticação.

Uma vez criados os usuários, grupos, participações, permissões e relacionamentos, o web2py irá prover uma *API* para verificar se um usuário está logado, se pertence a um grupo e se este grupo possui uma determinada permissão.

O web2py também provê *decoradores* de função que podem ser utilizados para restringir acesso a qualquer ação baseando-se no usuário, grupo e permissões.

O web2py também gerencia tipos específicos de permissões, as permissões que correspondem aos métodos CRUD (create, read, update, delete) e faz isso automaticamente, sem a necessidade do uso de decoradores.

Autenticação

Para utilizar o sistema RBAC, os usuários precisam estar identificados. Isso significa que eles precisam se registrar (ou serem registrados) para fazer o login.

Auth provê múltiplas formas de login. A padrão consiste em uma identificação baseada na tabela **auth_user**. Como alternativa, você pode autenticar seus usuários utilizando serviços de terceiros ou provedores de autenticação como, por exemplo, Google, PAM, LDAP, FaceBook, LinkedIn, OpenID, Oauth etc.

Para começar a utilizar **Auth**, você precisa definir no nível do **model** o seguinte código:

Note que este trecho de código já está definido na aplicação **welcome**, que serve de template para novas aplicações no web2py.

1. from gluon.tools import Auth
2. auth = Auth(globals(), db)
3. auth.define_tables(username=False)

Se você desejar utilizar o **username** ao invés do e-mail (padrão), atribua **True** no parâmetro **username**.

Para expor a ação de autenticação, por exemplo, no controlador **default.py**:

1. def user():
2. return dict(form=auth())

O web2py também possui uma view (default/user.html) padrão para renderizar a função de login:

1. {{extend 'layout.html'}}
2. <h2>{{=request.args(0)}}</h2>
3. {{=form}}
4. {{if request.args(0)=='login':}}
5. register

6. lost password

7. {{pass}}

Esta função simplesmente exibe um formulário de login e você pode customizá-la utilizando qualquer outro recurso do framework. A única condição é que o formulário a ser exibido depende do valor de **request.args(0)**, portanto, você sempre terá que manter um bloco **if** como este:

1. {{if request.args(0)=='login':}}...Formulário customizado...{{pass}}

O controlador e a view acima expõem multiplas ações:

[http://.../\[app\]/default/user/register](#)
[http://.../\[app\]/default/user/login](#)
[http://.../\[app\]/default/user/logout](#)
[http://.../\[app\]/default/user/profile](#)
[http://.../\[app\]/default/user/change_password](#)
[http://.../\[app\]/default/user/verify_email](#)
[http://.../\[app\]/default/user/retrieve_username](#)
[http://.../\[app\]/default/user/request_reset_password](#)
[http://.../\[app\]/default/user/reset_password](#)
[http://.../\[app\]/default/user/impersonate](#)
[http://.../\[app\]/default/user/groups](#)
[http://.../\[app\]/default/user/not_authorized](#)

- **register** permite que usuários se registrem, está integrado com um CAPTCHA que vem desabilitado por padrão.
- **login** permite que um usuário registrado efetue login.
- **logout** permite que o usuário efetue logout.
- **profile** permite que os usuários alterem informações de seu perfil (o conteúdo da tabela `auth_user`, que não possui estrutura fixa, e pode ser customizada).
- **change_password** permite alteração de senhas.
- **verify_email** caso a verificação via e-mail esteja habilitada, espera-se que o usuário receba um e-mail de confirmação de seu registro; este e-mail contém um link apontando para esta ação.
- **retrieve_username** permite que um usuário receba via e-mail seu nome de usuário informando seu e-mail.
- **request_reset_password** permite que um usuário requisiute a redefinição de sua senha caso tenha esquecido.
- **impersonate** permite que um usuário se passe por outro usuário, isto é útil para depuração e testes, e só é permitido caso o usuário possua a permissão *impersonate*.
- **groups** lista os grupos nos quais o usuário logado está contido.
- **not_authorized** exibe uma mensagem de erro caso o usuário tente acessar algo para o qual ele não tenha permissão de acesso.
- **navbar** é um helper que constrói dinamicamente os links para login, logout, etc.

As ações **logout**, **profile**, **change_password**, **impersonate** e **groups** requerem login para serem acessadas.

Por padrão tudo é exposto, porém, é possível restringir o acesso a apenas algumas dessas ações.

Todos os métodos mencionados acima podem ser estendidos ou substituídos através da herança da classe `Auth`.

Restringindo acesso

Para restringir acesso a uma função, permitindo apenas que usuários logados acessem, utilizamos o decorador `@auth.requires_login()`

1. `@auth.requires_login():`
2. `def segredo():`
3. `return dict(message='Bem vindo %(first_name)s' % auth.user)`
- 4.

`auth.user` é um dicionário contendo uma cópia dos registros de `db.auth_user` para o usuário que está atualmente logado. Há também o `auth.user_id`, que armazena o mesmo `auth.user.id` que armazena apenas o id do usuário.

Restrições no registro de usuários

Se você deseja que os visitantes registrem-se, mas só acessem o sistema após receberem aprovação do administrador, defina no **model**:

1. `auth.settings.registration_requires_approval = True`

Caso você queira restringir o acesso à página de registro:

1. `auth.settings.actions_disabled.append('register')`

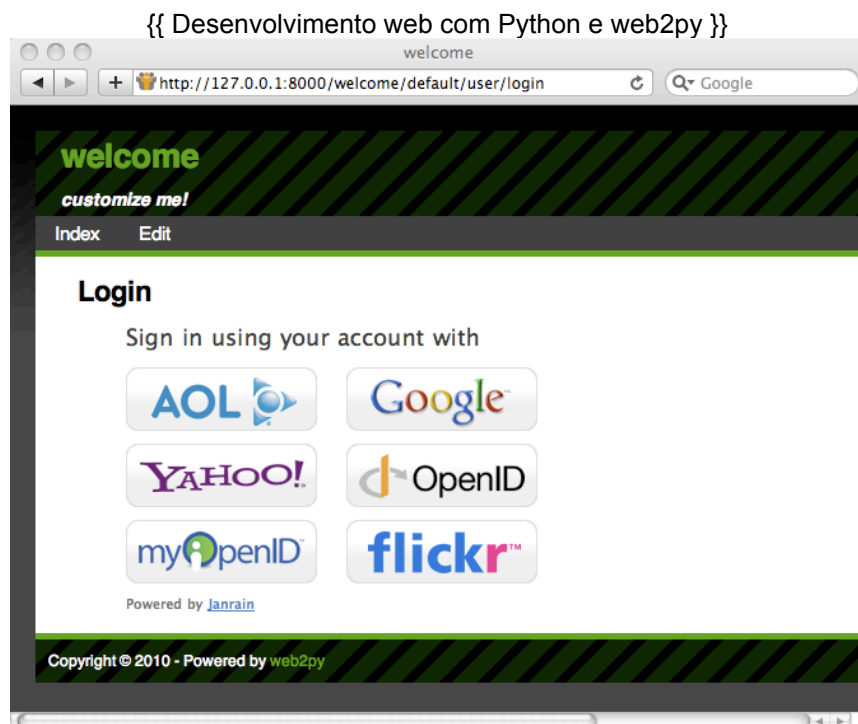
Integração com autenticação externa: Facebook, Google, Twitter etc.

Você pode continuar utilizando a camada de controle de acesso do web2py, porém utilizando serviços externos para efetuar a autenticação. O web2py fornece um módulo que implementa a autenticação com serviços como Google, Twitter, Facebook, LinkedIn, Myspace, Flickr etc.

A maneira mais fácil, e que já vem com implementação de exemplo da aplicação **welcome**, é utilizando o serviço (Janrain.com).

Janrain.com é um serviço que provê a ponte entre a sua autenticação e os provedores de assinatura. Você pode registrar-se para uma conta gratuita do janrain.com e utilizar da seguinte maneira:

1. `from gluon.contrib.login_methods.rpx_account import RPXAccount`
2. `auth.settings.actions_disabled=['register', 'change_password', 'request_reset_password']`
3. `auth.settings.login_form = RPXAccount(request,`
4. `api_key='...',`
5. `domain='...',`
6. `url = "http://localhost:8000/%s/default/user/login" % request.application)`



Quando um novo usuário fizer login no web2py através de um destes serviços, serão criados os registros necessários para que este usuário seja uma entidade do **Auth** e seu e-mail ficará armazenado na tabela **db.auth_user**. O web2py armazenará um **registration_id** baseado na identidade utilizada, para garantir que este usuário é um registro único.

Você pode customizar o mapeamento de dados fornecidos pelo jainrain.com com os dados de sua tabela de usuários **auth_user**. Exemplo para o Facebook:

1. `auth.settings.login_form.mappings.Facebook = lambda profile:\`
2. `dict(registration_id = profile["identifier"],`
3. `username = profile["preferredUsername"],`
4. `email = profile["email"],`
5. `first_name = profile["name"]["givenName"],`
6. `last_name = profile["name"]["familyName"])`

Se você preferir utilizar diretamente os meios de autenticação (Google, Facebook, OpenID, OAuth), você pode eliminar a necessidade do cadastro com o jainrain.com e implementar seu próprio sistema de autenticação customizado.

Autorização

Usuários pertencem a grupos. Cada grupo possui uma identificação. Grupos possuem permissões. Usuários possuem permissões herdadas dos grupos aos quais pertencem.

Você pode criar grupos, incluir usuários e permissões via **appadmin**: Considerando que possuímos um grupo chamado **RH** (Funcionários do depto de RH), podemos checar se o usuário logado atualmente possui associação a este grupo e desta maneira controlar o acesso a ações.

Decoradores

A maneira mais comum para checar as permissões de acesso é com o uso de decoradores de função:

```
1. def acao_publica():
2.     return 'essa função é pública'
3.
4. @auth.requires_login()
5. def acao_protegida():
6.     return 'esta função requer que o usuário esteja logado'
7.
8. @auth.requires_membership('RH')
9. def acao_protegida_porgrupo():
10.    return 'esta função requer que o usuário logado seja membro do grupo RH'
11.
12. @auth.requires(auth.user_id==1 or request.client=='127.0.0.1')
13. def acao_protegida_por_condicao():
14.    return 'esta função requer que o usuário tenha o id 1 e o IP 127.0.0.1'
```

Note que todas as funções, exceto a primeira, possuem restrições baseadas no login do usuário. Caso o usuário não esteja logado e as permissões não possam ser checadas, o web2py redireciona o usuário para a página de login.

Se um usuário logado não possuir a permissão de acesso a uma determinada ação, o web2py redireciona para uma página que pode ser configurada em **models** com:

```
1. auth.settings.on_failed_authorization = \
2.    URL('acao_a_direcionar')
```

Marcação de templates com Python

Assim como nos **models** e **controllers**, o web2py utiliza a própria linguagem Python para a marcação de template das views. Nas views é possível escrever qualquer tipo de código de marcação, como HTML, XML, CSS, JavaScript e embutir código Python no meio, e isso faz com que você tenha uma maior flexibilidade para gerar o conteúdo da maneira que quiser, podendo, por exemplo, programar o formato de saída de uma view em .csv, .rss ou até mesmo .xls do Microsoft Excel.

O web2py executa as views em uma cópia separada do ambiente de execução da aplicação. Para isso, transforma todo o código de marcação da view em um módulo com puro código Python - por este motivo, a execução da views é rápida.

- O código da view não precisa seguir regras de endentação do Python
- A marcação para escape é feita dentro de {{ e }}
- Blocos de código começam nas linhas {{{ terminadas em :
- Blocos de código terminam onde encontram a instrução {{pass}}
- Nos casos em que a construção do bloco for clara, não será preciso usar o {{pass}}

Apesar de o padrão MVC recomendar que se utilize as visões apenas para exibir conteúdo, com o web2py temos a **flexibilidade** de escolher como implementaremos nosso código, de qualquer forma, o web2py sempre encoraja o desenvolvedor a utilizar corretamente o padrão MVC e suas boas práticas.

Tudo o que está entre `{{...}}` será executado como código Python. Além disso, o web2py especifica algumas palavras-chave para ajudar no processo de criação de views. São elas:

- `{{=expressão_Python}}`: executa `expressão_Python` e imprime o resultado no local.
- `{{extend 'arquivo.html'}}`: utiliza a view `arquivo.html` como base para essa view.
- `{{include}}`: indica onde outras views que utilizam essa como base serão adicionadas.
- `{{pass}}`: marca o término de um bloco de código.

Pode parecer estranho, acima, o uso da keyword `pass`. Ela é usada por causa de uma definição no sistema de apresentação do web2py: por conta de espaços na camada de apresentação fazerem diferença, seria complicado em todos os casos o programador manter a endentação correta do código.

Para resolver esse problema o web2py então gera um novo código baseado em nossa view, ignorando a endentação que utilizamos - ele auto-endenta o código quando encontra `if`, `for` etc.

Para voltar um nível, como a endentação na view é ignorada, precisamos utilizar a palavra reservada do Python `pass`. Note que essa palavra reservada faz parte da linguagem Python e sua função é apenas "passar" (não executa comando algum) - nesse caso, ela serve como um sinal para o web2py voltar um nível na endentação.

escopo e parâmetros

Como as **views** são executadas em um ambiente paralelo ao ambiente de execução dos **controllers**, elas não têm acesso ao que é definido nas funções, ao menos que sejam variáveis de **sessão** ou o próprio retorno de uma função.

Mas as views têm acesso a todo o restante do escopo da aplicação, portanto qualquer função, objeto ou variável definidos nos **models** ficarão acessíveis para todas as views.

Desta mesma forma, o web2py já implementa alguns métodos próprios que podem ser utilizados tanto nas views, quanto nos controllers.

HTML Helpers

Considere o seguinte código em uma view:

```
1. {{=DIV('isto', 'é', 'um', 'teste', _id='123', _class='minhaclasse')}}
```

Que será renderizado como:

```
1. <div id="123" class="minhaclasse">istoéumteste</div>
```

DIV, por exemplo, é uma classe *helper*, um tipo de classe que pode ser usada para criar HTML programaticamente.

Em uma classe **helper**, os argumentos são interpretados como objetos contidos dentro da tag HTML. Os argumentos nomeados que iniciam com um *underline* serão interpretados como atributos. Alguns helpers recebem argumentos nomeados que não iniciam com *underline*; estes argumentos são específicos para cada tipo de TAG.

TAGS

A, B, BEAUTIFY, BODY, BR, CENTER, CODE, DIV, EM, EMBED, FIELDSET, FORM, H1, H2, H3, H4, H5, H6, HEAD, HR, HTML, I, IFRAME, IMG, INPUT, LABEL, LEGEND, LI, LINK, OL, UL, MARKMIN, MENU, META, OBJECT, ON, OPTION, P, PRE, SCRIPT, OPTGROUP, SELECT, SPAN, STYLE, TABLE, TAG, TD, TEXTAREA, TH, THEAD, TBODY, TFOOT, TITLE, TR, TT, URL, XHTML, XML, xmlescape, embed64

As tags podem ser agrupadas para gerar códigos HTML mais complexos:

```
1. {{=DIV(B(I("hello ", "<world>"))), _class="myclass')}}}
```

é renderizado como:

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

Document Object Model (DOM). Os helpers do web2py são muito mais do que um simples mecanismo para geração de HTML sem a necessidade de concatenação de strings. Eles provêm uma implementação do DOM no servidor. Objetos podem ser referenciados pela sua posição; exemplo:

```
1. >>> a = DIV(SPAN('a', 'b'), 'c')
2. >>> print a
3. <div><span>ab</span>c</div>
4. >>> del a[1]
5. >>> a.append(B('x'))
6. >>> a[0][0] = 'y'
7. >>> print a
8. <div><span>yb</span><b>x</b></div>
```

Atributos de um helper podem ser acessados através de seu nome, e o comportamento é semelhante ao de uma lista:

```
1. >>> a = DIV(SPAN('a', 'b'), 'c')
2. >>> a['_class'] = 's'
3. >>> a[0]['_class'] = 't'
4. >>> print a
5. <div class="s"><span class="t">ab</span>c</div>
```

XML, por exemplo, é um objeto usado para encapsular texto que não deve ser *escapado*, ou seja, será passado diretamente para o navegador. Este texto pode ou não conter XML válido, podendo ser código CSS ou Java Script.

```
1. >>> print DIV("<b>olá</b>")
2. &lt;b&gt;olá&lt;/b&gt;
```

Usando XML é possível prevenir o *escape*:

```
1. >>> print DIV(XML("<b>olá</b>"))
2. <b>olá</b>
```

SCRIPT inclui ou cria um link para um script JavaScript:

```
1. >>> print SCRIPT('alert("hello world");', _language='javascript')
2. <script language="javascript"><!--
3. alert("hello world");
4. //--></script>
```

Formulários

O HTML helper **FORM** é o responsável pela criação de formulários no web2py. Você pode utilizar FORM para criar formulários diretamente na **view**, ou criá-los no **controller** e então enviar o formulário como resposta para a view.

O helper FORM apenas cria uma TAG HTML **<form>....</form>**, e como todos os outros helpers, possui inteligência para conhecer os elementos inseridos dentro dele.

▪ Criação de formulário no controller utilizando FORM

O método **append** insere novos elementos em FORM.

```
1. def index():
2.     form = FORM(_action='', _method='post')
3.     form.append(LABEL('nome'))
4.     form.append(INPUT(_name='nome', type='text'))
5.     form.append(BR())
6.     form.append(LABEL('curso'))
7.     form.append(INPUT(_name='curso', _type='text'))
8.     form.append(BR())
9.     form.append(INPUT(_value='enviar', _type='submit'))
10.    return dict(form=form)
```

Criamos o formulário utilizando o helper **FORM** e armazenamos no objeto **form**, então retornamos um dicionário contendo **form** que será visível pela **view**.

O seguinte código na respectiva **view** já será suficiente para testar este formulário:

```
1. {{extend 'layout.html'}}
2. <br>
3. {{=form}}
4. {{=BEAUTIFY(request.vars)}}
```

{{=form}} renderiza o formulário, enquanto **{{=BEAUTIFY(request.vars)}}** formata e exibe o resultado do post deste formulário.

nome

curso

curso	:	python
nome	:	Bruno

- Criação de formulário diretamente na view

Em uma **view** você obviamente poderia criar o formulário utilizando as próprias tags HTML **<form>**, porém, às vezes é muito útil utilizar os **HELPERS** mesmo diretamente na view.

Considere o **controller**

```
1. def index():  
2.     return dict()
```

E então na view **default/index.html**

```
1. {{extend 'layout.html'}}  
2.  
3. {{=FORM(LABEL('nome'), INPUT(_name='nome', type='text'), BR(), LABEL('curso'),  
4.     INPUT(_name='curso', _type='text'), BR(), INPUT(_value='enviar', _type='submit'),  
5.     _action='', _method='post')}}  
6.  
7. {{=BEAUTIFY(request.vars)}}
```

Validação de formulários

Os validadores de formulário são os mesmos discutidos no capítulo [Tipos de dados e Validações](#), porém, no caso dos formulários, não consideramos os validadores do nível do banco de dados.

Alterando nosso primeiro exemplo, podemos incluir a validação em seus campos:

- No controller altere a função `index()`

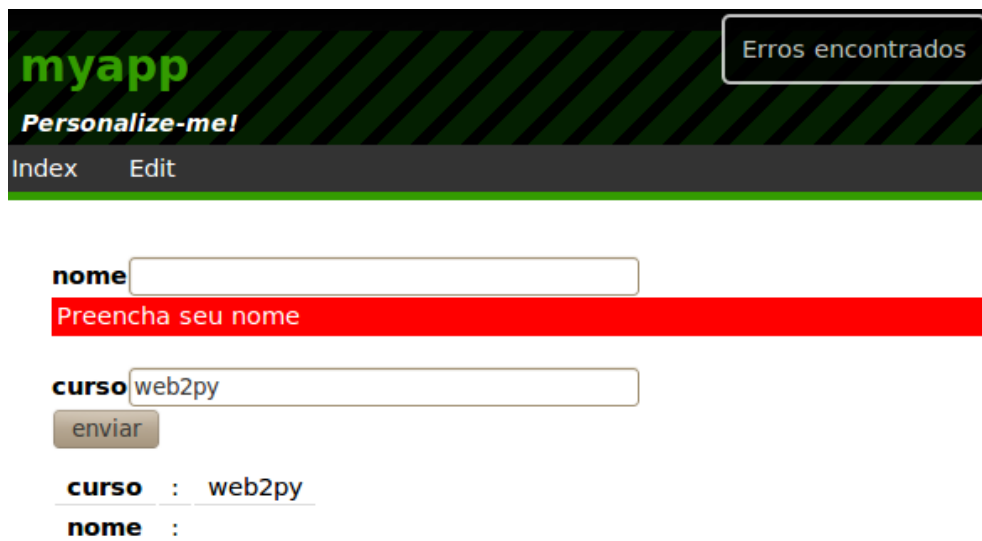
```
1. def index():
2.     # criação de uma DIV para servir de container
3.     container = DIV(_class='container')
4.
5.     # criação do formulário
6.     form = FORM(_action='', _method='post')
7.     form.append(LABEL('nome'))
8.     form.append(INPUT(_name='nome',
9.                       requires=IS_NOT_EMPTY(error_message='Preencha seu nome'),
10.                      type='text'))
11.     form.append(BR())
12.     form.append(LABEL('curso'))
13.     form.append(INPUT(_name='curso', _type='text'))
14.     form.append(BR())
15.     form.append(INPUT(_value='enviar', _type='submit'))
16.
17.     # inclusão do formulário dentro da DIV container
18.     container.append(form)
19.
20.     # validação do formulário
21.
22.     if form.accepts(request.vars, session):
23.         response.flash = 'Sucesso'
24.         container = DIV(H3('Dados enviados com sucesso! Obrigado'), _class='container')
25.     elif form.errors:
26.         response.flash = 'Erros encontrados, tente novamente'
27.
28.     return dict(container=container)
```

- Inserimos um objeto container, contendo uma `<div>` criada com o HTML helper `DIV`
- No campo `*nome*` do formulário incluímos o validador `IS_NOT_EMPTY`
- Inserimos o objeto `form` dentro do objeto `container` -
> `<div><form>.....</form></div>`
- Incluímos um bloco condicional que valida o retorno do método `accepts`
- Caso o formulário seja aceito pela validação, reiniciamos o objeto `container` e incluímos uma mensagem no lugar do formulário
- Efetuamos o retorno do objeto `container` para ser inserido na view

A view **default/index.html**:

1. `{{extend 'layout.html'}}`
- 2.
3. `
`
- 4.
5. `{{=container}}`
6. `{{=BEAUTIFY(request.vars)}}`

Se o formulário não passar pela validação (por exemplo, se o campo **nome** estiver vazio), o validador **IS_NOT_EMPTY** exibirá uma mensagem:



myapp
Personalize-me!

Index Edit

nome

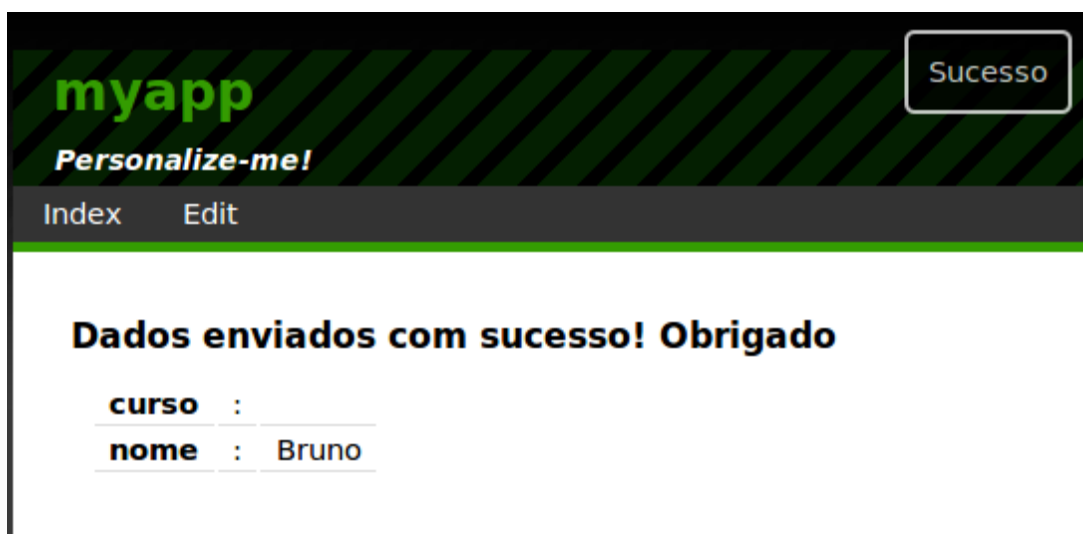
Preencha seu nome

curso

enviar

curso : web2py
nome :

Em caso de sucesso:



myapp
Personalize-me!

Index Edit

Sucesso

Dados enviados com sucesso! Obrigado

curso :
nome : Bruno

O web2py valida os formulários através do método **accepts** e garante que os dados não sejam inseridos em duplicidade, para isso cria uma variável de sessão contendo um id único identificando o formulário.

Criação de formulários automaticamente através do banco de dados

Esta é a forma mais interessante para a criação de formulários no web2py. Através de uma tabela do banco de dados, podemos criar formulários automaticamente:

Considerando o seguinte arquivo de **model**:

```
1. db = DAL('sqlite://storage.sqlite')
2. db.define_table('carro',
3.     Field('marca'),
4.     Field('modelo'),
5.     Field('foto', 'upload'))
6.
7. db.carro.marca.requires=IS_IN_SET(('Toyota','GM','Volvo', 'Renault'),
8. error_message='Insira um valor válido')
9. db.carro.modelo.requires=IS_NOT_EMPTY(error_message='Não pode ser em branco')
```

No **model** definimos uma tabela **carro**, possuindo os campos **marca** e **modelo**, cada um com seus validadores.

Repare que inserimos um campo chamado **foto** do tipo **upload** - este é um tipo especial para o web2py, e este campo servirá para armazenar o caminho de um arquivo que receberemos através de um campo de upload no formulário (os arquivos serão armazenados na pasta **/<aplicação>/uploads/carro.image.XXXX.jpg**).

Utilizando o helper SQLFORM, podemos criar um formulário de inserção de dados para esta tabela.

No arquivo de **controller** inclua:

```
1. def carro():
2.     # criamos um formulário a partir da tabela de carros
3.     form = SQLFORM(db.carro)
4.
5.     if form.accepts(request.vars, session):
6.         response.flash = 'Sucesso'
7.     elif form.errors:
8.         response.flash = 'Erros'
9.     else:
10.        response.flash = 'Preencha os dados'
11.
12.    return dict(form=form)
13.
```

Agora visite a seguinte URL: <http://127.0.0.1:8000/<aplicação>/default/carro>

Marca: Toyota ▼

Modelo:

Não pode ser em branco

Foto: Enviar arquivo...

Submit

O helper `SQLFORM` cria um formulário baseando-se nos campos e nos validadores de uma tabela. A validação final deste formulário pode ser feita na própria ação que está gerando o formulário utilizando o método **accepts**.

Download

Com o arquivo armazenado na pasta upload da aplicação, podemos exibi-lo utilizando a função **download** que já está definida no controller **default.py**

1. `def download():`
2. `return response.download(request, db)`

Como o caminho e o nome (que é gerado randomicamente para o arquivo que foi enviado) são armazenados na variável **request.vars.image**, podemos usá-la para gerar a visualização desta imagem.

Vamos criar uma nova ação que servirá para editar os dados de um carro existente na tabela **carro**:

1. `def mostracarro():`
2. `record = db.carro(request.args(0)) or redirect(URL('index'))`
3. `form = SQLFORM(db.carro, record, deletable=True,`
4. `upload=URL('download'))`
5. `if form.accepts(request.vars, session):`
6. `response.flash = 'Sucesso'`
7. `elif form.errors:`
8. `response.flash = 'Erros'`
9. `return dict(form=form)`

Repare que recebemos o id de um carro através do objeto **request** e na hora de criar o formulário com **SQLFORM**, passamos o nome da tabela **db.carro** e o registro referente ao id recebido. O argumento **deletable** determina se será possível excluir um registro através deste formulário e o argumento **upload** indica qual URL deverá ser usada para exibição da imagem; neste caso indicamos a função **download**.

Supondo que o id do carro que você inseriu seja 1, a página [http://.../aplicação >/default/mostracarro/1](http://.../aplicação/default/mostracarro/1) exibirá o seguinte formulário.

Id:	1
Marca:	Toyota ▼
Modelo:	Corolla
Foto:	<input type="text"/> Enviar arquivo... [file] <input type="checkbox"/> delete
	
Check to delete:	<input type="checkbox"/>
<input type="button" value="Submit"/>	

CRUD

Para facilitar ainda mais o trabalho com formulários, o web2py implementa uma API para o tratamento dos registros no banco de dados. Esta API oferece os métodos CRUD (create, read, update, delete).

É importante observar que CRUD é diferente de outros módulos do framework, pois precisamos importar explicitamente caso queiramos utilizar.

1. *# No arquivo de model*
- 2.
3. **from** gluon.tools **import** Crud
4. crud = Crud(globals(), db)
- 5.
6. *# passamos o banco de dados db como parâmetro*

Funções CRUD

- crud.tables() retorna a lista de tabelas do banco de dados.
- crud.create(db.tablename) retorna um formulário para criação de novos registros.
- crud.read(db.tablename, id) retorna um formulário somente leitura.
- crud.update(db.tablename, id) retorna um formulário de atualização do registro (id).
- crud.delete(db.tablename, id) exclui o registro (id).
- crud.select(db.tablename, query) retorna uma listagem de registros da tabela.
- crud.search(db.tablename) retorna um formulário de busca.
- crud() retorna uma das funções acima baseando-se em request.args().

Agora podemos criar no controller uma ação para executar as operações do CRUD:

```
1. def dados():
2.     return dict(form=crud())
3.
```

Com esta simples ação o web2py, através do **form=crud()**, irá expor as seguintes URLs:

- [http://.../\[app\]/\[controller\]/\[ação\]/tables](http://.../[app]/[controller]/[ação]/tables)
- `app/default/dados/tables`

[auth_user](#)

[auth_group](#)

[auth_membership](#)

[auth_permission](#)

[auth_event](#)

[carro](#)

- [http://.../\[app\]/\[controller\]/\[ação\]/select/\[nomedatabela\]](http://.../[app]/[controller]/[ação]/select/[nomedatabela])
- `app/default/dados/select/carro`

form :

carro.id	carro.marca	carro.modelo	carro.foto
1	Toyota	Corolla	file

- [http://.../\[app\]/\[controller\]/\[ação\]/read/\[nomedatabela\]/\[id\]](http://.../[app]/[controller]/[ação]/read/[nomedatabela]/[id])
- `app/default/dados/read/carro/1`

Marca: [Toyota](#)

Modelo: [Corolla](#)

Foto:



E seguindo o mesmo modelo expõe todas as ações CRUD

- [http://.../\[app\]/\[controller\]/\[ação\]/update/\[nomedatabela\]/\[id\]](http://.../[app]/[controller]/[ação]/update/[nomedatabela]/[id])
- [http://.../\[app\]/\[controller\]/\[ação\]/delete/\[nomedatabela\]/\[id\]](http://.../[app]/[controller]/[ação]/delete/[nomedatabela]/[id])
- [http://.../\[app\]/\[controller\]/\[ação\]/search/\[nomedatabela\]](http://.../[app]/[controller]/[ação]/search/[nomedatabela])

Para implementar validação e controle de permissões das funções CRUD, basta definir a variável **crud.settings.auth = None** e criar as permissões baseadas em registros.

JQuery

A aplicação de modelo do web2py **welcome** já vem com a biblioteca **JQuery**, incluindo os plugins **JQuery calendar**, **SuperFish Menus** e muitas funções customizadas baseadas em **JQuery**.

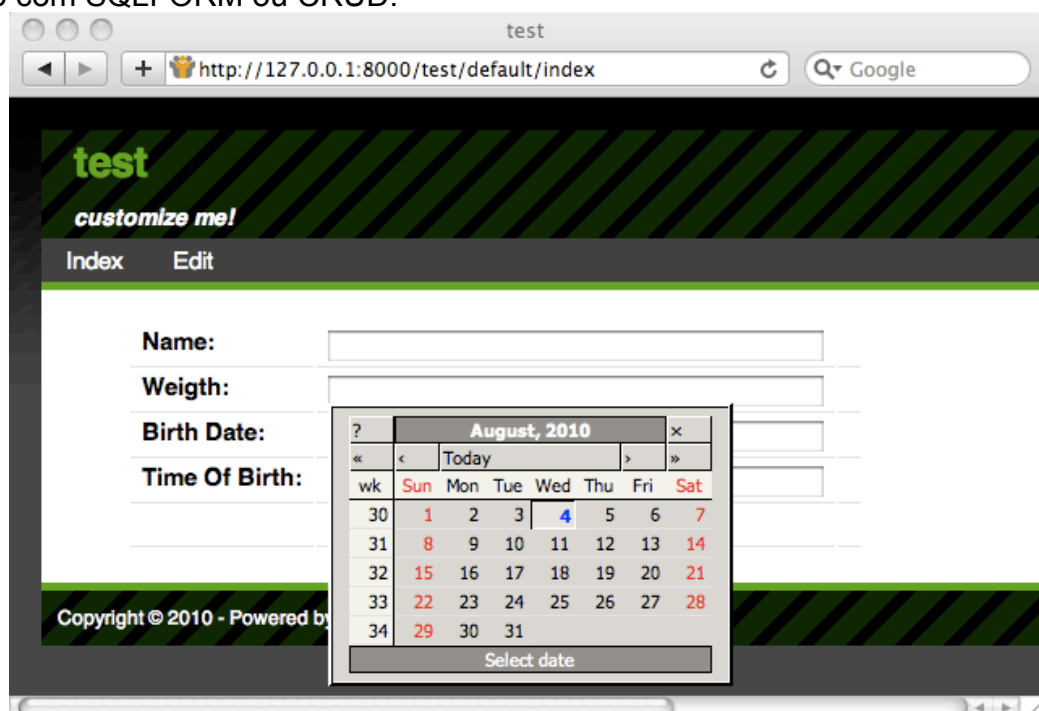
Você pode utilizar qualquer outra biblioteca javaScript, como: Prototype, ExtJS, Mochkit etc.

No layout da aplicação modelo é incluído um arquivo com várias funções JS:

1. view/web2py_ajax.html

Quando o web2py cria os formulários e seus validadores, utiliza as funções JQuery deste arquivo e os arquivos CSS definidos em layout.html para decorar os formulários.

Exemplo de JQuery Calendar em um campo do tipo datetime em um formulário criado com SQLFORM ou CRUD:



Funções JQuery podem ser incluídas nos Helpers. Exemplo:

```
1. {{=DIV('clique aqui !', _onclick="jQuery(this).fadeOut()")}}
```

O código acima cria uma div que ao ser clicada desaparece lentamente.

Ajax

A função **ajax** definida no arquivo `web2py_ajax.html` é uma função que efetua uma chamada a uma URL assincronamente. Esta função sempre retorna uma resposta que poderá ser embutida na sua página sem a necessidade de atualização.

A função **ajax** segue a seguinte sintaxe:

```
1. ajax(url, [id1, id2, ...], target)
```

Exemplo de implementação:

```
1. def repetemeunome():
2.     form = FORM(INPUT(_id='nome', _name='nome', _onkeyup="ajax('executacomajax',
3.     ['nome'], 'target')"))
4.     target = DIV(_id='target')
5.     return dict(form=form, target=target)
6.
7. def executacomajax():
    return str(request.vars.nome + ' ') * 10
```

A função **repetemeunome** criará um formulário com um campo **nome**. Neste campo, a cada caractere digitado, a função **executacomajax** será invocada e o retorno desta função será incluído na div **target**. Tudo isso acontece dinamicamente sem a necessidade de atualização da página.

form	:	<input type="text" value="Bruno"/>
target	:	Bruno Bruno Bruno Bruno Bruno Bruno Bruno Bruno Bruno Bruno

web2py na web

Onde encontrar ajuda:

1. **Grupo de usuários no google:**
 - a. <http://groups.google.com/group/web2py-users-brazil>
 - b. <http://groups.google.com/group/web2py>
2. **Site comunitário brasileiro:**
 - a. <http://www.web2pybrasil.com.br>
 - b. <http://web2pybrasil.appspot.com>
 - c. <http://twitter.com/web2pybrasil>

Onde contribuir:

1. **Rede de desenvolvedores voluntários:**
 - a. <http://openhatch.org/+projects/web2py>
 - b. <http://web2py.uservoice.com/forums/42577-general>
 - c. <http://www.web2pybrasil.com.br>

Recursos:

- **Aplicações e exemplos:**
 - <http://www.web2pyslices.com>
 - <http://web2py.com/appliances>

Instrutores:

Bruno Cezar Rocha @rochacbruno

Desenvolvedor e consultor web. Possui mais de 9 anos de experiência em desenvolvimento web e já trabalhou com as linguagens C, PHP, ASP, C#. Atualmente, ministra treinamentos, palestras e desenvolve soluções com Python para web com os frameworks Pylons e web2py. É um dos colaboradores do site web2pybrasil, e também vezes escreve também no blog <http://rochacbruno.com.br>. Além dos projetos com Python, atua como desenvolvedor e coordenador de projetos (ScrumMaster) na empresa GENTE - <http://servicogente.com.br>

Contato:

<http://twitter.com/rochacbruno> - rochacbruno@gmail.com

Álvaro Justen @turicas

Graduando em Eng. de Telecomunicações pela UFF, onde desenvolve atividades de pesquisa, ensino e extensão. É desenvolvedor da Intelie e ativista de software livre há mais de 8 anos; colaborador no desenvolvimento do web2py; disseminador do Arduino e entusiasta de metodologias ágeis. Participante assíduo de eventos e grupos de usuários, palestra e organiza eventos, como PythOnCampus, Arduino/web2py Hack Day e Dojorio. Escreve no blog <http://blog.justen.eng.br>

Contato:

<http://twitter.com/turicas> - alvarojusten@gmail.com

Autoria: **Bruno Cezar Rocha**
Colaboração: **Álvaro Justen**
Revisão gramatical: **Claudia P.**

Esta apostila contém partes traduzidas do livro oficial do web2py (<http://web2py.com/book>), esta tradução foi devidamente autorizada pelo autor do livro, **Massimo di Pierro**.



Desenvolvimento web com Python e web2py by [Bruno Cezar Rocha](#) is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike License](#).

Based on a work at <http://web2py.com>

Permissions beyond the scope of this license may be available at web2pybrasil.com.br.