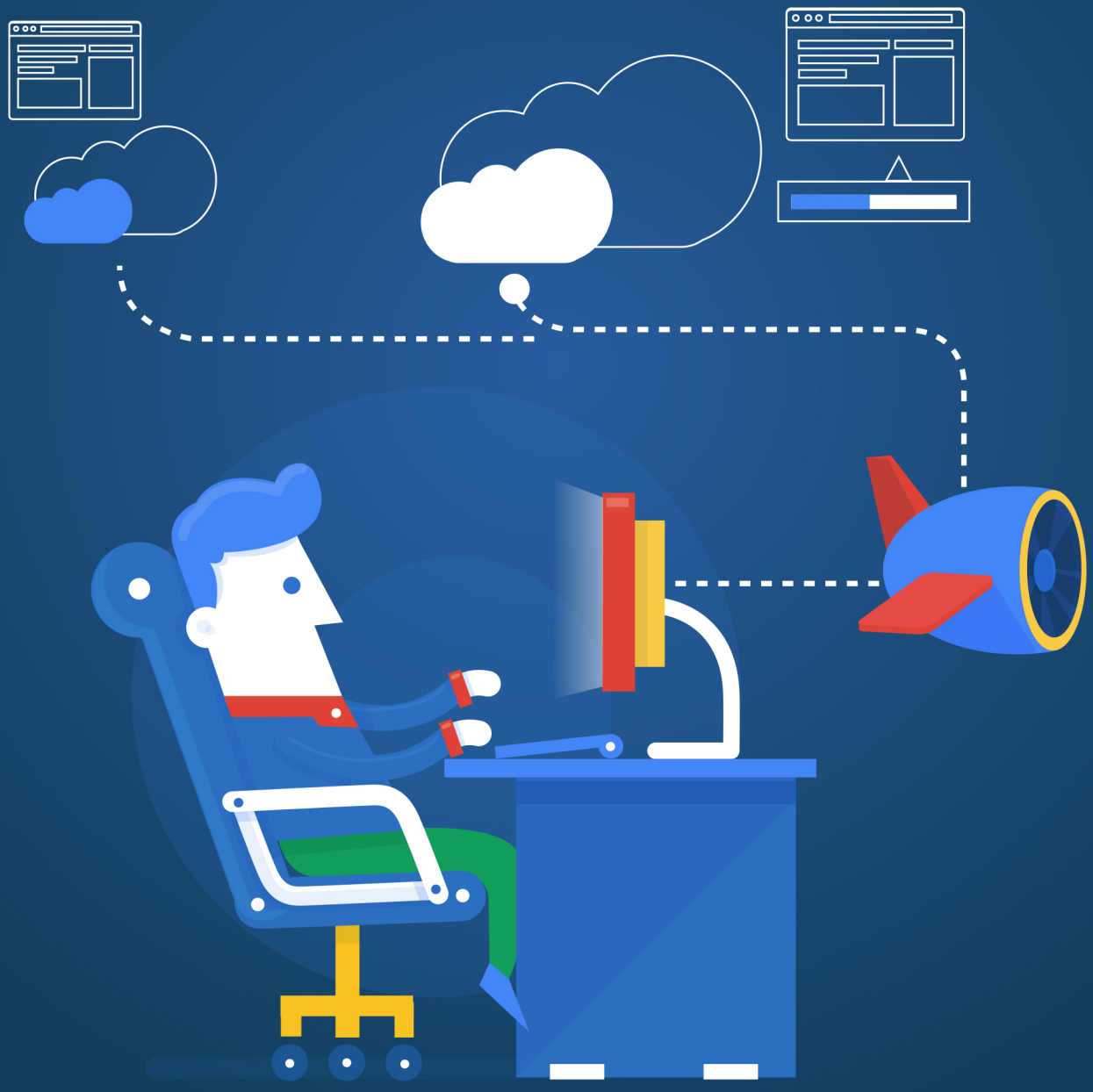
python pro

# APP ENGINE & PYTHON

## VOCÊ PROGRAMA E O **GOOGLE** ESCALA



RENZO NUCCITELLI

# App Engine e Python

## Você programa e o Google escala!

Renzo Nuccitelli

Esse livro está à venda em http://leanpub.com/appengine

Essa versão foi publicada em 2015-05-07


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Renzo Nuccitelli

# Tweet Sobre Esse Livro!

Por favor ajude Renzo Nuccitelli a divulgar esse livro no Twitter!

O tweet sugerido para esse livro é:

Confira esse livro fantástico sobre Aplicações WEB: App Engine e Python!

A hashtag sugerida para esse livro é #appengineescala.

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

https://twitter.com/search?q=#appengineescala

*Dedico esse livro à minha mãe Amanda e à minha noiva Priscila, que sempre me apoiam, por mais malucas que sejam minhas idéias.*

*Ao meu grande amigo Reginaldo Filho, vulgo "Beraba", que me apresentou o Google App Engine.*

*Aos meus amigos Dênis Costa e Giovane Liberato, que criaram e me permitiram usar o sensacional título desse livro.*

*Ao Luciano Ramalho, meu tutor e parceiro no site Python Pro.*

*A todos alunos, que me motivam a continuar ensinando.*

*A todos os mestres que passaram pela minha vida e me salvaram pela educação.*

# Conteúdo

# Prefácio

"A educação é a arma mais poderosa que você pode usar para mudar o mundo."
- *Nelson Mandela*

---

## Relação de Confiança

Durante minha vida passei por algumas situações[1] e precisei de ajuda. E por isso fui salvo pela educação. Se você não pode pagar, envie um email para renzo.n@gmail.com para poder adquirir o livro gratuitamente de forma legal.

Em contrapartida peço o seguinte, na base da confiança:

1. Curta a fan page[2];
2. Divulgue o livro entre seus amigos;
3. Envie sua opinião após a leitura, permitindo sua divulgação;
4. Compre uma cópia quando estiver em condições.

## Por que Google App Engine (GAE)?

No início de 2010 topei o desafio de desenvolver um site destinado ao tráfego de fotos. Eu já desenvolvia software, mas não tinha conhecimento profundo sobre todo o processo. Em particular, não conhecia nada sobre servidores e linux.

Então comecei a procurar um *host*. Montei meu ambiente de desenvolvimento. Contudo, achava que a estrutura escolhida não suportaria o volume esperado de fotos. Era época da explosão dos sites de compras coletivas e meus clientes pretendiam fazer promoções neles.

Foi então que num almoço meu amigo Reginaldo me apresentou a solução: Google App Engine (GAE). Para me convencer, fez um "Hello World" em 5 minutos, incluindo o deploy. Aliado à simplicidade, vinha a promessa de que o site escalaria automaticamente.

Depois de 3 meses de desenvolvimento em minhas horas livres, estava pronto o Revelação Virtual, precursor do Pic Pro[3]. Após 2 meses de testes, veio a prova de fogo: foi executada uma promoção no

[1] http://blog.renzo.pro.br/2013/10/quando-voce-quer-o-universo-conspira-em.html
[2] https://www.facebook.com/pythonappengine
[3] http://www.picpro.com.br

Groupon. A promessa foi cumprida e o GAE aguentou todos picos de transferência de arquivos. Até hoje o sistema é utilizado e já trafegou mais de 2 milhões de fotos.

Mas nem tudo foi fácil. Apesar da simplicidade, a plataforma exigiu de mim uma mudança de paradigma. E como toda mudança, levou certo tempo para me acostumar.

Mas apesar das diferenças, as vantagens se mostraram maiores que os problemas. E é isso que mostrarei. Abordarei não só conceitos e exemplos simples, mas apresentarei soluções para problemas reais e recorrentes no mundo das aplicações web.

# Por que Python?

À época do projeto supracitado eu era fluente apenas em Java. Por conta disso, iniciei o projeto nessa linguagem. Contudo o GAE era muito diferente do que estava acostumado. Não permitia o uso de vários frameworks consagrados, como o Hibernate[4].

Resolvi então testar a linguagem Python, a primeira suportada pela plataforma. Mais do que resolver o problema de desenvolvimento, me apaixonei pela linguagem e por sua comunidade. Ela me permitiu ser mais expressivo, exigindo a escrita de muito menos código. Portei o Pic Pro para Python em 1 semana. A quantidade de código diminuiu 65%.

Desde então trabalho sempre com Python, tanto para desenvolvimento de aplicações comerciais como para ensinar. Ela é excelente para iniciantes que desejam aprender programação de uma maneira prática, simples e expressiva.

# Público

Este livro foi escrito para quem deseja aprender a fazer uma aplicação completa para internet. Mais do que isso, ele é ideal para quem quer transformar uma ideia em um produto web rapidamente, sem ter que aprender a complexa arte de escalar servidores.

A documentação do GAE é excelente, objetiva e centralizada. Contudo, muitas vezes apresenta ferramentas superficialmente. Sendo assim, esse livro é uma excelente fonte para você ir além do básico, desenvolvendo soluções profissionais e robustas.

# Conhecimento Prévio

É recomendável o conhecimento básico sobre protocolo HTTP, HTML, Javascript e CSS. Se você não possui algum deles, pesquise cada assunto somente quando necessário para entender os exemplos.

[4] http://hibernate.org/

# Código Fonte e Vídeo Aulas

Todo código fonte contido no livro é livre e pode ser encontrado em https://github.com/renzon/appengineepython. Além disso vídeos aulas gratuitas podem ser encontradas em https://www.youtube.com/playlist?list=PLA05yVJtRWYRGIeBxag8uT-3ftcMVT5oF

# Organização do Conteúdo

O conteúdo está organizado nos seguintes capítulos:

1. Ambiente de Desenvolvimento: como montar o ambiente local de desenvolvimento;
2. Webapp2: apresentação do *framework* web *Webapp2*;
3. Tekton: apresentação do *framework* web *Tekton*;
4. Frontend: linguagem de template e arquivos estáticos;
5. Usuários e Segurança: Login de usuários, segurança e controle de permissões;
6. Banco de Dados: persistência de dados utilizando o Big Table;
7. Arquitetura de Apps: arquitetura para camada de negócios com *GaeBusiness*;
8. AJAX com AngularJS: chamadas AJAX com uso do framework AngularJS;
9. Agendamento, Fila de Tarefas e Email: fila e agendamentos de tarefas para processamento e envio email;
10. Serviços Remotos: acesso a serviços de outros sites (*web services*);
11. Upload e Download: upload e download de arquivos;
12. Testes Automatizados: como testar automaticamente uma aplicação.

Todos capítulos contém questões e respectivas respostas em seu final para fixação do conhecimento.

O material aqui produzido é resultado de muito estudo, prática e dedicação. Divirta-se!

# Ambiente de Desenvolvimento

"Se quiser derrubar uma árvore em metade do tempo, passe o dobro do tempo amolando o machado."
- *Provérbio chinês de autor desconhecido*

---

## Introdução

O ambiente de desenvolvimento é o conjunto de ferramentas que o desenvolvedor utiliza para construir software em seu computador. Ambientes complexos, com erros ou que demoram a executar suas tarefas comprometem profundamente a produtividade. Por isso, como no caso do machado da parábola, mantenha-o sempre "afiado".

Nesse capítulo serão instalados os itens necessários para o desenvolvimento: interpretador Python, Kit de Desenvolvimento de software (SDK, do inglês *Software Development Kit*). Além desses, também será utilizado um Ambiente Integrado de Desenvolvimento (IDE, do inglês *Integrated Development Enviroment*), o Pycharm.

## Interpretador Python

Atualmente existem duas principais versões da linguagem: 2.7 e 3.4. Sempre é bom utilizar a última, por conta das melhorias. Mas muitos frameworks ainda não foram portados para a nova versão. O App Engine é um caso e por isso a plataforma só aceita a versão 2.7 da linguagem Python.

No Linux (Ubuntu versão 12.04) elá já está disponível por padrão. Para instalar em outros sistemas, visite a página de downloads[5] e escolha a versão adequada ao seu Sistema Operacional.

Para verificar aquela que está instalada em seu sistema, abra um terminal de comando e digite `python`. A versão disponível é impressa como primeira linha após a execução, conforme exemplo a seguir:

[5] http://www.python.org/getit/

**Execução do Python via linha de comando no Ubuntu 12.04**

```
Python 2.7.3 (default, Sep 26 2013, 20:03:06)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Caso apareça uma mensagem informando que o comando não existe, deve ser instalado o interpretador através do *link* já mencionado. Se a instalação já foi feita, verifique se o endereço de instalação foi adicionado ao *path*.

# Terminal de Comando

Para ser possível emular e desenvolver a aplicação em um computador, é necessário instalar o Kit de Desenvolvimento Padrão[6] (SDK, do inglês *Standard Development Kit*). Ele provê as ferramentas necessárias para inicialização do servidor localmente, interação com banco de dados, entre outras.

Assim como o interpretador, o endereço de instalação também deve ser adicionado ao *path*. Para fazer isso no Linux, acesse a sua pasta *home* e utilize o atalho "Crtl+h" para visualizar os arquivos ocultos. Dentre esses arquivos se encontram dois de interesse: o ".bashrc" e o ".profile". Edite esses arquivos adicionando em seus finais as seguintes linhas:

**Adição de variável de Ambiente no Linux**

```
export GAE_SDK="seu diretorio"/google_appengine
PATH=$PATH:$GAE_SDK
```

A seguir se encontra um exemplo para um SDK que foi instalado em diretório *bin* localizado na pasta *home*:

**Arquivo .bashrc editado para incluir SDK ao path**

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
                . /etc/bash_completion
fi

export GAE_SDK=/home/renzo/bin/google_appengine
PATH=$PATH:$GAE_SDK
```

Cabe ressaltar que a variável `GAE_SDK` não é obrigatória. Contudo, ela será utilizada no capítulo de Testes Automatizados. E por ser parte da configuração, ela já foi aqui inserida.

[6] https://developers.google.com/appengine/downloads?hl=pt-br

## “Olá Mundo” - Terminal

Dentro da pasta do Kit de Desenvolvimento existe um diretório contendo um modelo de projeto. Acesse-o e copie a pasta “new_project_template” para um endereço de sua preferência. A figura 1.01 a seguir mostra o conteúdo do SDK após extração do arquivo zipado:

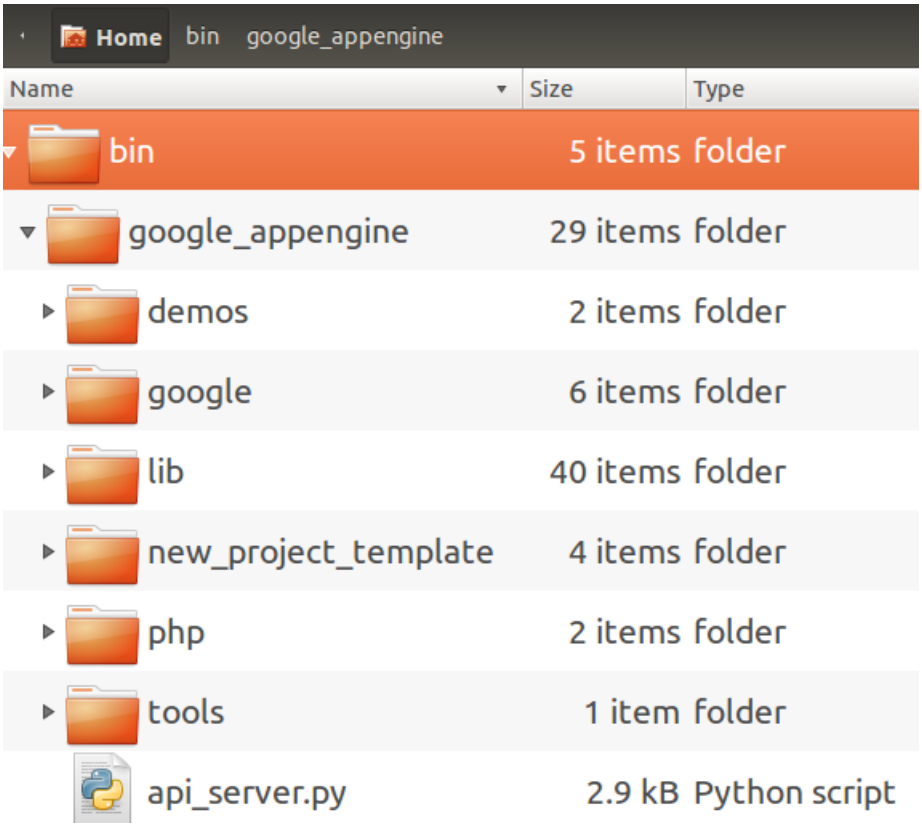



**Figura 1.01: Conteúdo SDK GAE**

Copiada a estrutura do projeto, navegue até seu diretório no terminal e digite o seguinte comando `dev_appserver.py .` para iniciar o servidor localmente.

A figura 1.02 mostra o console no caso de execução correta do programa. A partir desse momento o SDK está servindo a aplicação localmente. Acesse o endereço http://localhost:8080 em seu navegador e confira se aparece a mensagem “Hello World”. Em caso negativo, confira os passos anteriores. Desligue o servidor utilizando o atalho “Ctrl+C”.

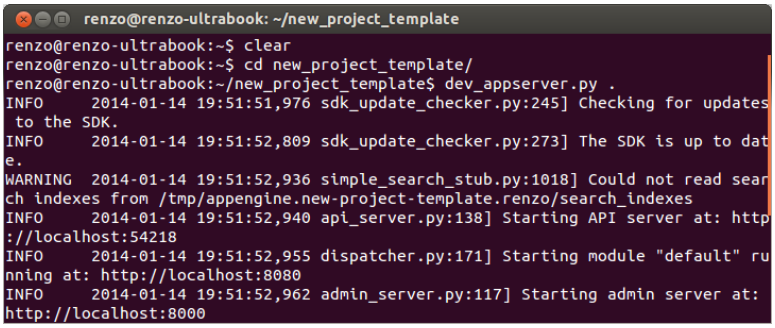



**Figura 1.02: Comando dev_appserver.py**

Observe que a primeira opção depois do comando é o diretório onde se encontra o arquivo de configuração app.yaml. Sendo assim, também funcionaria o comando `dev_appserver.py new_project_template` se executado direto do diretório home.

# Pycharm

Apesar de ser possível utilizar a linha de comando e um editor de texto simples para desenvolver, nesse livro será utilizado o Ambiente de Desenvolvimento Integrado (IDE, do inglês *Integrated Development Enviroment*) Pycharm[7]. Outros também podem ser utilizados , como o Pydev[8], que inclusive é gratuito. Mas aquele foi escolhido por ser robusto e por seu conjunto de ferramentas.

Para conseguir executar o Pycharm é necessário instalar a Máquina Virtual Java[9] (JVM, do inglês *Java Virtual Machine*). Após baixar o pacote adequado ao seu sistema também é necessário adicioná-lo ao *path*. As seguintes linhas devem ser adicionadas aos arquivos .bashrc e .profile, onde a variável JAVA_HOME deve apontar para o diretório onde foi extraído o JDK:

**Arquivo .bashrc editado para colocar JDK no path**

```
export JAVA_HOME=/home/renzo/bin/jdk1.7.0_45
PATH=$PATH:$JAVA_HOME/bin
```

Cumpridas essas etapas, navegue até o diretório bin da pasta de instalação do Pycharm no terminal e rode o script pycharm.sh. Exemplo de execução do comando `~$ ./bin/pycharm-3.0.1/bin/pycharm.sh` . Com o ambiente inicializado você pode fixar seu ícone na barra de tarefas para poder inicializar a aplicação apenas com um clique.

Prefira baixar a versão profissional do Pycharm, pois a versão gratuita não possui alguns plugins, como o destinado a construção e aplicações Google App Engine.

[7] http://www.jetbrains.com/pycharm/download/
[8] http://pydev.org/
[9] http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk7-downloads-1880260.html

## “Olá Mundo” - IDE

Ao iniciar a IDE, clique em *Create New Project* e escolha o tipo de projeto *Google App Engine project*. Você deve assinalar um identificador para o seu projeto e informar o diretório onde se encontra o SDK do Google App Engine. A tela de criação de projeto deve ficar semelhante à figura 1.03:

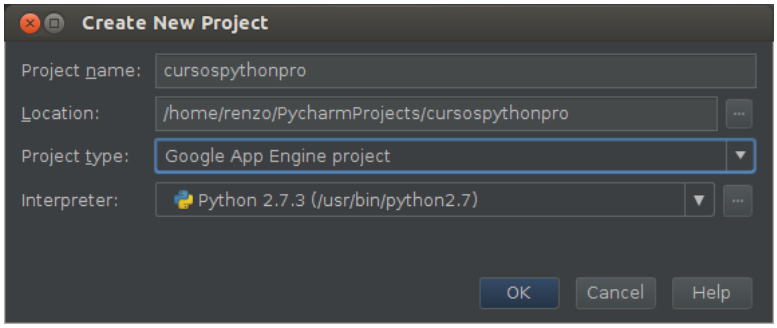



**Figura 1.03: Novo projeto GAE - Pycharm**

Para executar o servidor local, basta clicar no ícone verde *play* que se encontra em destaque na figura 1.4.

Verique se no link `http://localhost:8080` , utilizando seu navegador, aparece a mensagem “Hello World”. Em caso negativo, confira e repita os passos anteriores.

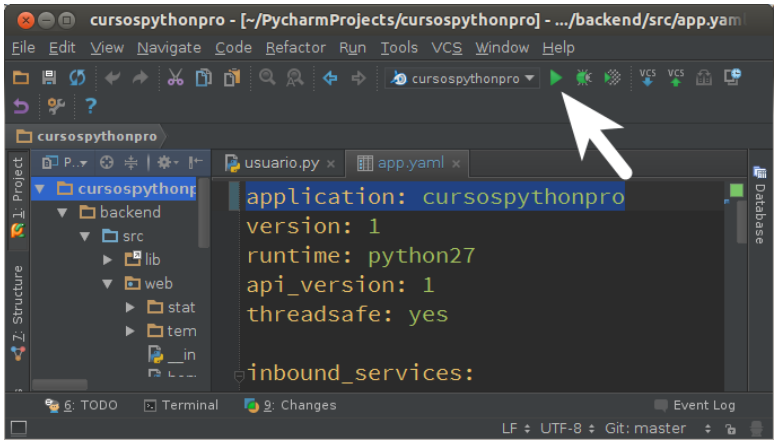



**Figura 1.04: Iniciando servidor local - Pycharm**

# Publicação (*Deploy*)

Muitos programadores iniciantes começam a desenvolver suas aplicações logo após aprenderem como inicializar o servidor localmente. Contudo, o processo de publicação do site é um passo fundamental. Somente dessa maneira os usuários podem acessá-lo pela rede.

O *deploy* deve ser simples. Em um bom processo de desenvolvimento, deve ser possível publicar apenas com um clique ou comando. Mais do que isso, o site deve ser publicado de maneira

frequente. Por essa razão, recomenda-se iniciar o desenvolvimento apenas após a publicação do site pela primeira vez, mesmo que seja apenas para imprimir uma mensagem simples. Afinal de contas, não faz sentido desenvolver um site se não for possível disponibilizá-lo. Como afirmaram Jez Humble e Dave Farley em seu livro Entrega Contínua: "Atualmente muitas companhias estão fazendo múltiplos *deploys* em apenas um dia"[10]. Foge ao escopo desse livro implementar o processo automatizado. Contudo, fazer o *deploy* manualmente, antes do desenvolvimento em si, é o primeiro passo nesse sentido.

## Painel de Controle

Para publicação, é necessário a criação de uma aplicação no Painel de Controle do Google App Engine. Isso deve ser feito no endereço http://appengine.google.com. Será necessário utilizar uma Conta Google. Após o cadastro, será possível visualizar o painel.

Clicando "Create Application" uma nova aplicação é gerada. Um identificador único e um nome devem ser atribuídos à aplicação, como na figura 1.05:

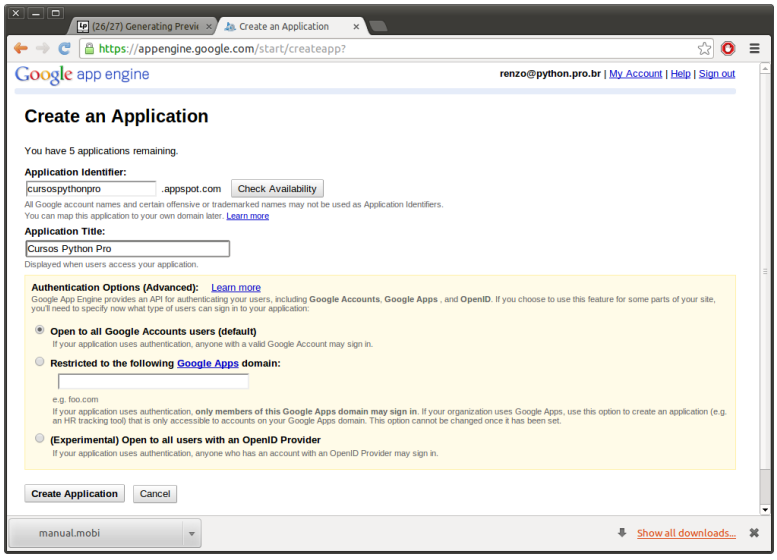



**Figura 1.05: Formulário App Engine**

Cadastrada a aplicação, o arquivo de configuração app.yaml que se encontra na raiz de seu projeto deve ser editado. O identificador deve ser copiado na primeira linha, no item "application". Um exemplo da primeira linha do arquivo se encontra a seguir:

**Primeira linha do arquivo app.yaml**

```
application: cursospythonpro
```

[10] "These days, many companies are putting out multiple releases in a day". *Continuous Delivery*, p. xxiii

Para usar o App Engine a conta Google precisa ser verificada por celular através de um sms. Usuários já se queixaram dessa mensagem nunca chegar para clientes da operadora Oi. Para evitar isso, pode ser utilizado um celular de outra operadora. Outra opção é pedir para alguém já cadastrado enviar um convite, via painel de administração, para administração de um projeto já existente. Nesse caso, a verificação não é necessária.

## *Deploy* - Terminal de Comando

Editado o arquivo app.yaml, deve ser acessado o diretório do projeto no terminal e digitado o comando "update" do script appcfg.py do SDK:

**Deploy via bash**

```
~$ cd PycharmProjects/cursospythonpro/
/PycharmProjects/cursospythonpro$ appcfg.py update .
```

O programa pedirá e-mail e senha. Devem ser inseridos os dados relativos à conta que se utilizou para criar a aplicação no painel de controle. Com esse comando o SDK irá escanear os arquivos do projeto e enviá-los ao servidor.

Finalizada a publicação, é possível acessar o site através do endereço **http://seu_identificador.appspot.com**, conforme imagem 1.06:

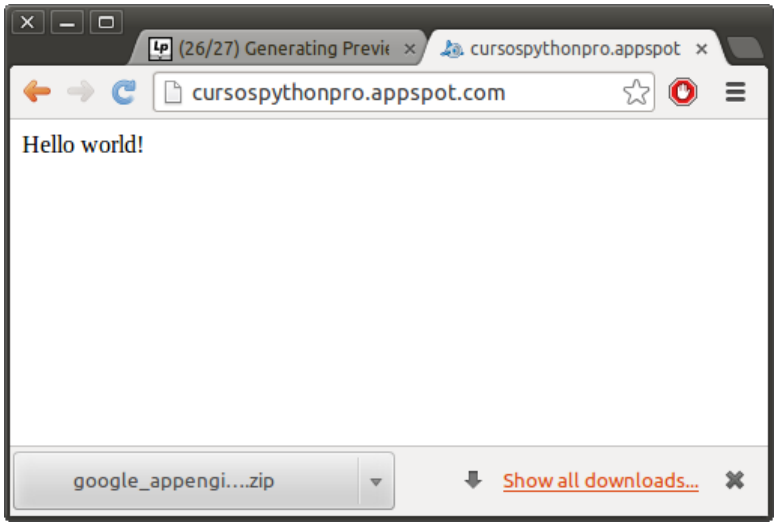



**Figura 1.06: Site publicado no App Engine**

Para quem usa login em dois passos[11], deve ser utilizado o comando `appcfg.py update . --oauth2`. Será exibida a tela de *login* no navegador onde é possível inserir a senha e o código de segurança.

[11] http://www.google.com/intl/pt-PT/landing/2step/

## *Deploy* via Pycharm

Para fazer o deploy com o Pycharm, o menu *Tools* > *Google App Engine* > *Deploy Application* deve ser acessado, conforme figura 1.07:

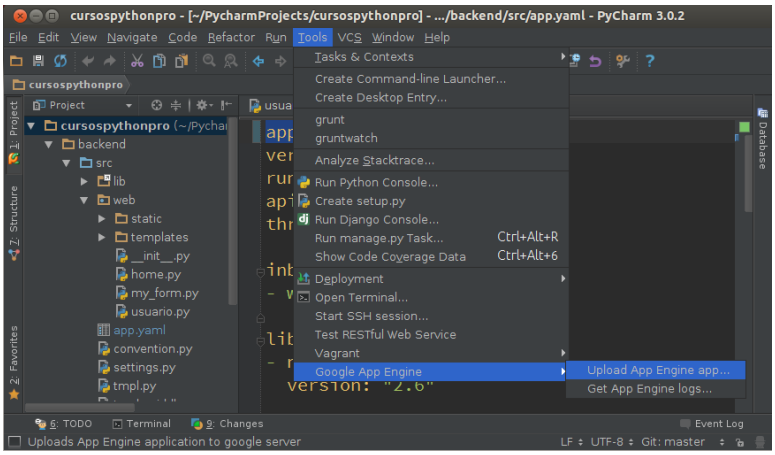



**Figura 1.07: Deploy via Pycharm**

Devem ser inseridas as credenciais. Finalizado o processo, é possível acessar o site no endereço supracitado.

Para quem usa login em dois passos[12], a opção "Use Passwordless login via OAuth2" de ser escolhida. Será exibida a tela de *login* no navegador onde é possível inserir a senha e o código de segurança.

# Windows e Mac

o SDK oferece uma Interface Visual (GUI, do inglês *Graphical User Interface*) que pode ser utilizada em opção à linha de comando para os Sistemas Operacionais Windows Mac OS. Clicando em em *File* > *Add new Application* um novo projeto é criado, como na figura 1.08:

[12] http://www.google.com/intl/pt-PT/landing/2step/

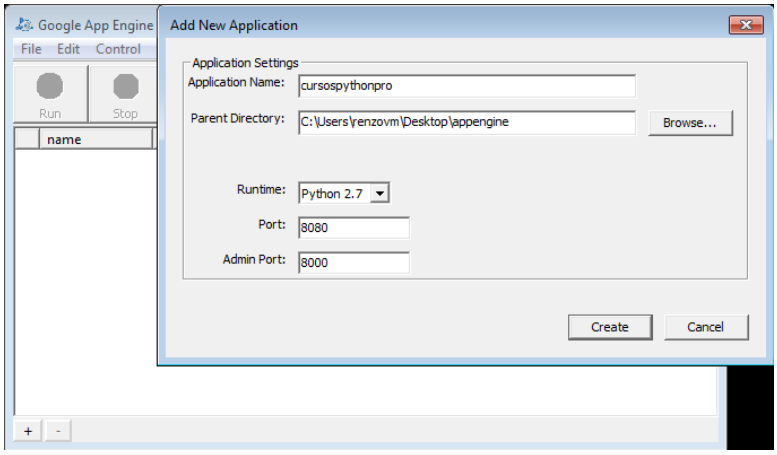



**Figura 1.08: GUI do App Engine**

Criado o projeto é possível iniciar o servidor local clicando no botão *Run* e fazer o *deploy* clicando no botão com esse nome. É importante notar que a utilização da IDE Pycharm faz com que o desenvolvimento ocorra da mesma forma nos diferentes sistemas operacionais. Por isso ele será utilizado como base para os exemplos.

Outra fator interessante é que ao instalar no Mac, ele informa que criará um link simbólico para a pasta do SDK. Esse link se encontra em */usr/local/google_appengine* e deve ser utilizado quando necessário para referenciar as bibliotecas da plataforma.

# Resumo

Nesse seção foi explicado como instalar as ferramentas necessárias para executar o servidor localmente: Interpretador Python versão 2.7 e SDK do Google App Engine. Além disso, foi instalado o Pycharm, IDE que facilita o desenvolvimento.

Mais do que a instalação, foi ensinado como executar o servidor localmente e como fazer a publicação do projeto via linha de comando, IDE e também via GUI do SDK do Google App Engine para Windows e Mac.

Esse conhecimento possibilitará o desenvolvimento e testes nos próximos capítulos.

# Questões

1. Quais as duas principais versões atuais do interpretador Python?
2. Qual versão de Python é suportada atualmente pelo App Engine?
3. Para que serve o SDK do App Engine?
4. Qual a vantagem de colocar ferramentas instaladas no *path* do Sistema Operacional?
5. O que é e para que serve uma IDE?
6. Como se chama o arquivo de configuração de uma aplicação App Engine?
7. Qual o domínio padrão para acessar aplicações App Engine na internet?

# Respostas

1. As duas principais versões atuais do interpretador Python são 2.7 e 3.4.
2. A versão de Python suportada atualmente pelo App Engine é a 2.7.
3. O SDK do App Engine serve para acessar as ferramentas de desenvolvimento, como servidor local e banco de dados.
4. A vantagem de colocar ferramentas no *path* do Sistema Operacional é poder executar comandos independente do diretório em que se encontre o Terminal de Comando.
5. IDE é um Ambiente Integrado de Desenvolvimento. Ele serve para fornecer ferramentas que facilitam o desenvolvimento e faz com que o processo de desenvolvimento seja independente de sistema operacional.
6. O arquivo de configuração de uma aplicação App Engine se chama app.yaml
7. O domínio padrão para acessar aplicações App Engine na internet é appspot.com

# Webapp2

"Linguagem não é simplesmente um dispositivo para relatar experiências, mas um *framework* que as define." [13]
- *Benjamin Whorf*

---

## Introdução

A palavra *framework* significa um esquema, um conjunto de passos, que serve para resolver determinado problema. Apesar do termo ser geral, ele é muito utilizado em computação como sinônimo de biblioteca. Ou seja, um conjunto de códigos que se utilizados facilitam a construção de um sistema.

Por conta disso é importantíssimo saber quais são as questões que um *framework* busca resolver. Se alguém pedisse para uma pessoa se vestir a caráter, a pergunta óbvia seria: "Qual a ocasião?". Sendo assim, seguir os passos de uma biblioteca sem saber seu objetivo é análogo a ir vestido de fraque em uma partida de futebol.

Nesse capítulo será explicado o funcionamento do *framework* Webapp2 e seu objetivo.

## O que é Webapp2?

Webapp2 é uma biblioteca de código aberto utilizada na documentação oficial introdutória ao GAE. Ela implementa o padrão WSGI (*Web Server Gateway Interface*) e pode ser utilizada em outras plataformas que forneçam integração com esse padrão.

Não será ela a biblioteca base utilizada para construir a maior parte dos exemplos nesse livro. Contudo, seu entendimento é fundamental por duas razões:

1. O *framework* Tekton irá utilizá-la como base;
2. Algumas vezes é necessário fazer uso do Webapp2, utilizando objetos como 'Request', 'Response' e 'Handler'.

A seguir constam seções explicando os diferentes componentes dessa ferramenta.

[13] "Language is not simply a reporting device for experience but a defining framework for it."

# Arquivo app.yaml

No projeto criado no capítulo anterior existe um arquivo de configuração chamado `app.yaml` conforme listagem 2.01:

**Listagem 2.01: Arquivo app.yaml**

```
application: new-project-template
version: 1
runtime: python27
api_version: 1
threadsafe: yes

libraries:
- name: webapp2
  version: "2.5.2"

handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: .*
  *script*: main.app
```

Esse arquivo contém as principais informações sobre o projeto. Sempre que se quiser entender sua estrutura geral, deve-se verificar o conteúdo desse arquivo, que será detalhado nas próximas seções.

## Cabeçalho Inicial

No cabeçalho inicial do arquivo se encontram informações básicas sobre o projeto, conforme listagem 2.02:

**Listagem 2.02: Cabeçalho**

```
1 application: new-project-template
2 version: 1
3 runtime: python27
4 api_version: 1
5 threadsafe: yes
```

Na linha 1, *application*, consta o identificador da aplicação. Conforme foi visto na seção *deploy* do capítulo anterior, esse código deve ser o mesmo utilizado na criação da aplicação no console do GAE. Através dele o SDK identifica o projeto e consegue publicar o site corretamente na nuvem.

Já na linha 2 consta a versão da aplicação. É importante notar que o GAE permite a existência de várias versões simultâneas. A figura 2.01 mostra uma aplicação com múltiplas versões:

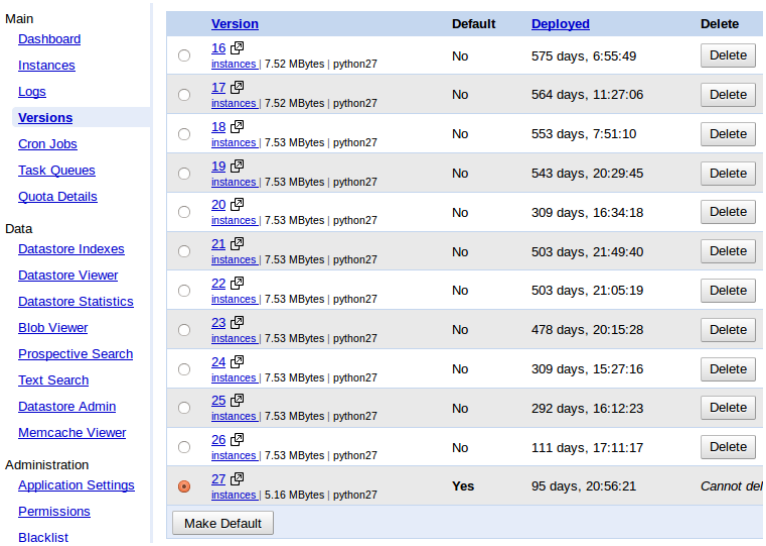



**Figura 2.01: Múltiplas Versões**

Nesse exemplo a versão 27 é padrão e, portanto, ela serve a aplicação quando acessada através do endereço http://picprolabs.appspot.com. Para acessar diferentes versões, como a 16 por exemplo, seu número deve ser concatenado ao início do domínio. Uma primeira forma de fazer isso é acessar o endereço http://16-dot-picprolabs.appspot.com/. Outro endereço válido é http://16.picprolabs.appspot.com/. Recomenda-se utilizar o primeiro, pois ele evita problemas no caso de acesso seguro via https.

Cabe ressaltar que essa funcionalidade é muito útil para se testar a aplicação antes de torná-la disponível aos clientes. Ou seja, publica-se o site em uma versão específica, alterando a versão no arquivo de configuração. Suas funcionalidades são conferidas e homologadas no respectivo endereço dedicado. Em caso de sucesso, a nova versão é definida como padrão, ficando disponível a todos usuários. Outra vantagem é poder retornar à versão anterior em caso de problemas. E tudo isso é feito com apenas um clique no painel de administração.

As demais linhas definem a versão da linguagem e api. Além disso, informam se uma instância da aplicação pode processar requisições em paralelo.

## Bibliotecas Embutidas

O GAE fornece um conjunto de bibliotecas embutidas. Para instalação, deve ser editada a seção *libraries* do arquivo de configuração, conforme listagem 2.03.

Cada item define o nome da biblioteca a ser instalada e, opcionalmente, sua respectiva versão. No exemplo foi instalada a versão 2.5.2 da biblioteca Webapp2.

**Listagem 2.03: Instalação do framework Webapp2**

```
1 libraries:
2 - name: webapp2
3   version: "2.5.2"
```

## Roteamento via Arquivo de Configuração

Roteamento é uma questão a ser resolvida por qualquer *framework web*. É ele quem define qual código será executado no servidor de acordo com o *path* acessado no navegador. O início do roteamento se dá na definição da seção *handlers*, no arquivo de configuração, conforme listagem 2.04:

**Listagem 2.04: Roteamento**

```
1 handlers:
2 - url: /favicon\.ico
3   static_files: favicon.ico
4   upload: favicon\.ico
5
6 - url: .*
7   *script*: main.app
```

Nas linhas 2 a 4 é definido um *handler* para servir o arquivo estático `favicon.ico` no *path* `/favicon.ico`. Esse endereço padrão é utilizado pelo navegador para colocar uma imagem respectiva ao site na aba em que ele está aberto, conforme figura 2.02:

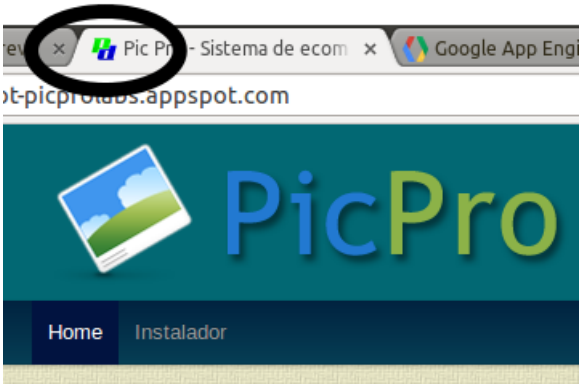



**Figura 2.02: favicon.ico**

Mas o assunto de arquivos estáticos será melhor abordado no capítulo 4, *Frontend*. Nesse, o foco é a execução de código Python que ocorre quando um endereço é digitado no navegador.

Na linha 6, contendo a expressão *url*, estão sendo mapeados todos os *paths* através da expressão regular ".*". Essa expressão se traduz como: "qualquer cadeia de caracteres". Cabe ressaltar que os *handlers* são processados na ordem em que aparecem. Por essa razão o primeiro trata a chamada em `/favicon`, enquanto o segundo irá tratar todos os demais.

Uma vez mapeado o *path*, deve ser informado qual será o *script* que deverá processar a requisição. Isso é feito na linha 7, onde é configurado o arquivo 'main.py'. Outros *handlers* poderiam ser adicionados ao arquivo para processar outros endereços.

# Roteamento via Webapp2

Uma vez que o arquivo de configuração aponta para um *script*, é importante entender seu conteúdo, que se apresenta na listagem 2.05 a seguir:

**Listagem 2.05: Script main.py**

```
1  # -*- coding: utf-8 -*-
2  from __future__ unicode_literals
3  import webapp2
4
5
6  class HomeHandler(webapp2.RequestHandler):
7      def get(self):
8          self.response.write('Olá Mundo!')
9
10
11 app = webapp2.WSGIApplication([('/', HomeHandler)],
12                                debug=True)
```

Na linha 3 é importado o módulo webapp2 pertencente ao *framework* de mesmo nome. Como se quer construir um *handler* para tratar requisições, é construída a classe `HomeHandler` herdando de `RequestHandler` na linha 6. Nessa classe foi sobrescrito o método referente ao respectivo verbo HTTP. No caso do código, sobrescreveu-se o método `get` na linha 7, referente ao verbo HTTP GET.

Por fim, é muito comum em Python termos várias classes declaradas em um módulo. Sendo assim, apenas acrescentar o *script* no arquivo de configuração não é suficiente para se saber qual *handler* deve ser executado. Por essa razão é necessário fazer também o roteamento dentro do código, como consta na linha 12. Nela é criado o parâmetro `app` construindo-se um objeto do tipo `WSGIApplication`.

O primeiro parâmetro da construção é uma lista de tuplas, onde o primeiro argumento é a expressão regular mapeando os *paths*. No exemplo, está sendo mapeada a raiz do projeto `'/'`. Já o segundo parâmetro indica a classe, `HomeHandler`, que irá ser utilizada para tratar a requisição.

Com esse código escrito e depois de executar o servidor, é possível obter a mensagem "Olá Mundo" no navegador, conforme a figura 1.03:

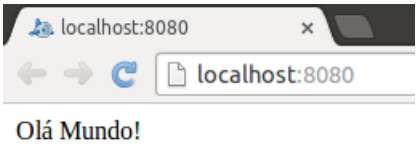



**Figura 2.03: Mensagem "Olá Mundo!" no navegador**

Seguindo a mesma filosofia, o *script* `main.py` pode ser editado para responder "Olá Wepapp2!" quando se acessa o *path* `/outra`. As mudanças se encontram no Código 2.01:

**Código 2.01: Script main.py**

```
1  class HomeHandler(webapp2.RequestHandler):
2      def get(self):
3          self.response.write('Olá Mundo!')
4
5
6  class OutroHandler(webapp2.RequestHandler):
7      def get(self):
8          self.response.write('Olá Wepapp2!')
9
10
11 app = webapp2.WSGIApplication([('/', HomeHandler)], debug=True)
12 app = webapp2.WSGIApplication([('/', HomeHandler),('/outra', OutroHandler)],
13                                debug=True)
```

Na figura 2.04 é exibido o resultado do acesso ao *path* no navegador:

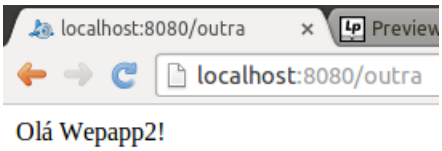



**Figura 2.04: Mensagem "Olá Webpp2!" no navegador**

Dessa maneira, os passos para se fazer o roteamento são:

1. Configurar o arquivo app.yaml toda vez que se deseja criar um *script* Python;
2. Configurar cada *Handler* dentro de seu respectivo *script* com o Webapp2.

# Request

Quando o usuário acessa um site no navegador, ele está enviando uma requisição HTTP. O Webapp2 processa essa requisição, construindo um objeto do tipo `Request`. É através dessa interface que o código do servidor obtêm acesso às informações e parâmetros enviados pelo usuário.

Parâmetros podem ser enviados através do que se chama *query string*, que é parte da url localizada após o sinal "?". Sendo assim, quando se faz uma chamada HTTP do tipo GET, é possível editar os valores enviados modificando a url no navegador. Na figura 2.05 consta um exemplo onde são passados os parâmetros `nome` com valor `Renzo` e `sobrenome` com valor `Nuccitelli`. Cada parâmetro é divido utilizando-se o sinal "&":

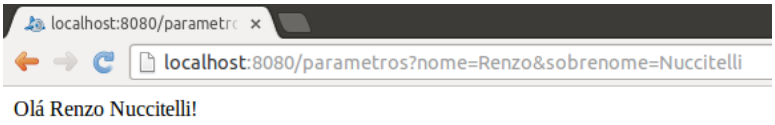



**Figura 2.05: Url com query string" no navegador**

É importante notar que acessando a url, a mensagem apresentada não é mais estática. Ela é construída com base nos parâmetros enviados. Para isso se utiliza o método `get` do objeto `Request`, fornecendo o nome do parâmetro do qual se deseja extrair o valor. Os valores obtidos das requisições sempre são do tipo `string`. O código 2.02 apresenta o handler com os métodos `get`, nas linhas 3 e 4, em destaque:

**Código 2.02: Handler com extração de parâmetros**

```
1 class ParametrosHandler(webapp2.RequestHandler):
2     def get(self):
3         nome = self.request.get('nome')
4         sobrenome = self.request.get('sobrenome')
5         self.response.write('Olá %s %s!' % (nome, sobrenome))
```

Cabe ressaltar que se o parâmetro inspecionado não estiver presente na *query string*, o método `get` irá retornar `None` como valor.

Muitas outras informações podem ser extraídas do objeto, tais como *cookies*, cabeçalhos HTTP e domínio. Mas esses outros métodos serão vistos no decorrer do livro, sendo a obtenção de parâmetros o foco nesse momento.

# Response

Após o recebimento de uma requisição, o servidor deve enviar uma resposta utilizando protocolo HTTP. Para cumprir esse objetivo, é utilizado o objeto do tipo `Response`. Ele provê métodos para auxiliar no envio de dados.

Nos exemplos anteriores seu método `write` foi utilizado para enviar uma `string` como resposta às requisições. O código 2.03 contém lógica que se utiliza desse método, na linha 5, em destaque:

**Código 2.03: Método write para escrita de strings**

```
1 class ParametrosHandler(webapp2.RequestHandler):
2     def get(self):
3         nome = self.request.get('nome')
4         sobrenome = self.request.get('sobrenome')
5         self.response.write('Olá %s %s!' % (nome, sobrenome))
```

De forma semelhante ao `Request`, o objeto `Response` possui métodos para se alterar *cookies* e cabeçalhos HTTP. Esses métodos também serão vistos nos próximos capítulos.

# Redirect

Muitas vezes ao se acessar uma url o usuário é redirecionado para outra. Isso ocorre com frequência após a submissão de um formulário. Seu objetivo é evitar que a requisição para salvamento de dados seja enviada novamente, caso o navegador tenha seu botão de atualizar pressionado.

Para executar esse redirecionamento, o objeto `RequestHandler` fornece o método `redirect`. A ele deve ser fornecido como parâmetro a url completa, no caso de um servidor externo. No caso de um endereço interno da aplicação, apenas o *path* precisa ser utilizado. O código 2.04 contém código exemplificando os dois casos:

**Código 2.04: Método redirect**

```
1  class RedirecionaParaOutroHandler(webapp2.RequestHandler):
2      def get(self):
3          self.redirect('/outra')
4
5
6  class GoogleHandler(webapp2.RequestHandler):
7      def get(self):
8          self.redirect(str('http://www.google.com'))
9
10
```

```
11 app = webapp2.WSGIApplication([('/', HomeHandler),
12                                 ('/outra', OutroHandler),
13                                 ('/redirecionar', RedirecionaParaOutroHandler),
14                                 ('/google', GoogleHandler),
15                                 ('/parametros', ParametrosHandler)],
16                                debug=True)
```

Dessa maneira, ao acessar `http://localhost:8080/redirecionar`, o usuário será redirecionado para `http://localhost:8080/outra`. Por outro lado, se acessar `http://localhost:8080/google`, será redirecionado para `http://www.google.com`.

É importante ressaltar que em um redirecionamento é enviada uma resposta HTTP de código 30x. Portanto, existe tráfego de dados durante essa operação. Devem ser evitados múltiplos redirecionamentos consecutivos, pois a maioria dos navegadores não permitem mais do que 5 redirecionamentos encadeados. O objetivo é evitar o redirecionamento infinito e consumo excessivo de recursos de rede.

# Resumo

Nesse capítulo foi apresentado um resumo do *framework* de código aberto Webapp2. Através de seus 3 principais objetos, `Request`, `Response` e `RequestHandler` é possível obter dados dos usuários, enviar informações do servidor e redirecioná-los para outros endereços.

Mais do que simplesmente utilizar essa biblioteca, foi importante entender que ela serve para abstrair o protocolo HTTP. Sendo assim, precisamos apenas conhecer seus componentes para construir um *web site*. Apesar de simples, os poucos componentes vistos são suficiente para construirmos toda a navegação de uma aplicação.

O Webapp2 não será o *framework* base para a construção dos exemplos desse livro. Mas seu conhecimento é fundamental, pois é com base nele que irá funcionar o *framework* Tekton, que será o assunto do próximo capítulo.

# Questões

1. Qual o nome do arquivo de configuração do Google App Engine?
2. Para que serve o item `application` do arquivo de configuração?
3. Para que serve o item `version` do arquivo de configuração?
4. Qual endereço deve ser utilizado para acessar uma aplicação com id `foo` e versão `35`?
5. Para que serve a seção `libraries` do arquivo de configuração?
6. Para que serve a seção `handlers` do arquivo de configuração?
7. Como são definidos os *paths* mapeados no arquivo de configuração?
8. Por que é necessário mapear `RequestHandlers` nos *script*s Python?
9. Para que serve a classe `RequestHandler`?
10. Como se relacionam os métodos da classe `RequestHandler` e os do protocolo HTTP?
11. Para que serve o objeto `Request`?
12. Como se obtém os valores de parâmetros enviados via *query string* em uma chamada HTTP do tipo GET?
13. Para que serve o objeto `Response`?
14. Qual o método do objeto `Response` serve para enviar *strings*?
15. Como é possível enviar uma resposta para redirecionamento?

# Respostas

1. O nome do arquivo de configuração do Google App Engine é app.yaml.
2. O item `application` serve para identificar a aplicação. Ele deve conter o mesmo id definido no momento da criação da aplicação no painel de controle disponível em http://appengine.google.com.
3. O item `version` serve para identificar qual versão da aplicação será utilizada no momento da publicação do site.
4. O endereço para acessar a aplicação `foo` em sua versão `35` deve ser `http://35.foo.appspot.com.br` ou `http://35-dot-foo.appspot.com.br`. É recomendado utilizar a segunda forma para evitar problemas quando o acesso for feito via https.
5. A seção `libraries` serve para configurar as bibliotecas a serem utilizadas na aplicação. O GAE fornece um conjunto de bibliotecas que podem ser instaladas dessa maneira.
6. A seção `handlers` serve para mapear os *script*s Python que serão executados de acordo com o *path* das requisições HTTP.
7. Para definição dos *paths* são utilizadas expressões regulares.
8. É necessário mapear `RequestHandlers` nos *script*s Python porque é comum a definição de múltiplas classes em um arquivo. Sendo assim somente o mapeamento via arquivo de configuração não é suficiente para definir qual classe será utilizada para processar uma requisição.
9. A classe `RequestHandler` serve para definir o código que irá processar uma requisição HTTP.
10. Os métodos da classe `RequestHandler` devem ser sobrescritos para processar as chamadas HTTP de tipo com mesmo nome. Por exemplo, uma chamada HTTP do tipo GET será processada no método sobrescrito get, Uma chamada do tipo POST em um método post e assim or diante.
11. O objeto `Request` serve como interface para acesso às informações sobre uma requisição HTTP.
12. Os valores de parâmetros enviados via *query string* em uma chamada HTTP do tipo GET são obtidos utilizando-se o método `get` do objeto `Request`. A ele deve ser fornecido o nome do parâmetro do qual se quer extrair o valor.
13. O objeto `Response` server como interface para construção de uma resposta HTTP. Ela contém os dados a serem enviados como resposta a uma requisição.
14. O método *write* da classe `Response` serve para enviar *strings*.
15. Para enviar uma resposta de redirecionamento deve ser utilizado o método `redirect` do objeto `RequestHandler`, fornecendo como parâmetro o *path* desejado, no caso de redirecionamento para a mesmo aplicação, ou a url completa em caso contrário.

# Tekton

"Você deve renunciar toda superficialidade, toda convenção, toda presunção e desilusão." [14]
- *Gustav Mahler*

---

## Introdução

Convenção em vez de Configuração (*Convention over Configuration*) é o grande mantra desse capítulo. Será explicado o funcionamento do *framework* Tekton[15] e todas suas convenções. A ideia principal é evitar a excessiva configuração do sistema, permitindo o foco em funcionalidades.

Diferente da abordagem do capítulo anterior, Webapp2, o roteamento e recebimento de parâmetros será feito através de convenções. Apesar disso, o conhecimento sobre os objetos básicos, `Request`, `Response` e `RequestHandler`, será fundamental.

Tekton será a biblioteca base. Nela serão construídos todos os exemplos de código no restante do livro.

## Setup inicial

O repositório do Tekton consiste em um template para iniciar um novo projeto. É possível copiar esse código baixando o arquivo zipado[16]. Após a extração, ele apresenta a seguinte estrutura de pastas:

```
tekton-master
└── backend
    ├── appengine
    ├── apps
    ├── build_*script*s
    ├── test
    └── venv
```

A seguir são apresentadas breves descrições de cada um dos diretórios:

[14] "You must renounce all superficiality, all convention, all vanity and delusion."
[15] http://github.com/renzon/tekton
[16] https://github.com/renzon/tekton/archive/master.zip

1. backend: diretório raiz de todo código do servidor;
2. appengine: aqui se encontram todos arquivos de configuração e integração com o GAE;
3. apps: contém as aplicações, com suas lógicas de negócio, que compõem o sistema;
4. build_scripts: possui *scripts* para geração de produtos, como arquivos de internacioanalização;
5. test: diretório com *scripts* de testes;
6. venv: pasta que define o ambiente virtual do sistema.

Os detalhes sobre essas estruturas serão apresentados nas próximas seções.

# Virtualenv

Virtualenv é uma biblioteca Python. Ela permite que se crie um ambiente isolado para cada projeto. É possível então definir qual a versão da linguagem e quais são as bibliotecas externas que serão utilizadas.

Instruções de instalação para os sistemas Linux, Mac OS e Windows.

## Virtualenv Linux e Mac OS

A instalação do virtualenv no Linux pode ser feita através do comando `sudo apt-get install python-virtualenv`, sendo necessária a senha de `root` para instalação.

Já no Mac OS é necessário instalar a biblioteca utilizando o instalador de pacotes Python (pip). Para isso, basta instalar a versão 2.7 com o [Homebrew](http://docs.python-guide.org/en/latest/starting/install/osx/: `brew install python`.

Essa versão já inclui o pip em sua instalação. O comando `pip install virtualenv` deve então ser utilizado para instalar o virtualenv.

Depois de instalado o virtualenv, o *script* de setup do projeto é o mesmo para Linux e Mac. A pasta `venv` do projeto contém o *script* `venv.sh`. Ele cria o ambiente isolado e instala as bibliotecas necessárias. Um exemplo de execução do *script* é exibido a seguir:

**Execução do *script* venv.sh**

```
~/PycharmProjects/appengineepython$cd backend/venv/
~/PycharmProjects/appengineepython/backend/venv$./venv.sh
```

Como as dependências necessárias são baixadas da internet, é necessária conexão com a internet.

## Virtualenv Windows

Para instalar o virtualenv no Windows se faz necessário a instalação do pip, uma biblioteca python para instalação de pacotes. Para isso deve ser baixado o *script* Python get_pip.py[17]. Depois o *script* deve ser executado com o interpretador Python, conforme exemplo a seguir:

[17] https://raw.github.com/pypa/pip/master/contrib/get-pip.py

**Execução do *script* get_pip.sh**

```
C:\Users\renzovm>cd Desktop
C:\Users\renzovm\Desktop>python get_pip.py
Downloading/unpacking pip
Downloading/unpacking setuptools
Installing collected packages: pip, setuptools
Successfully installed pip setuptools
Cleaning up...
```

Após a instalação o pip deve ser adicionado ao *path*. Para isso deve ser aberto o explorer e utilizado o botão direito do mouse para clicar em `computador`. Depois deve ser acessado o menu `propriedades`, `Configurações Avançadas de Sistema`, e na porção `Variáveis do Sistema` a variável *Path* deve ser editada para conter em seu final o diretório de instalação do pip. No caso da instalação padrão, esse endereço é `C:\Python27\Scripts`. A figura 3.01 exemplifica o processo:

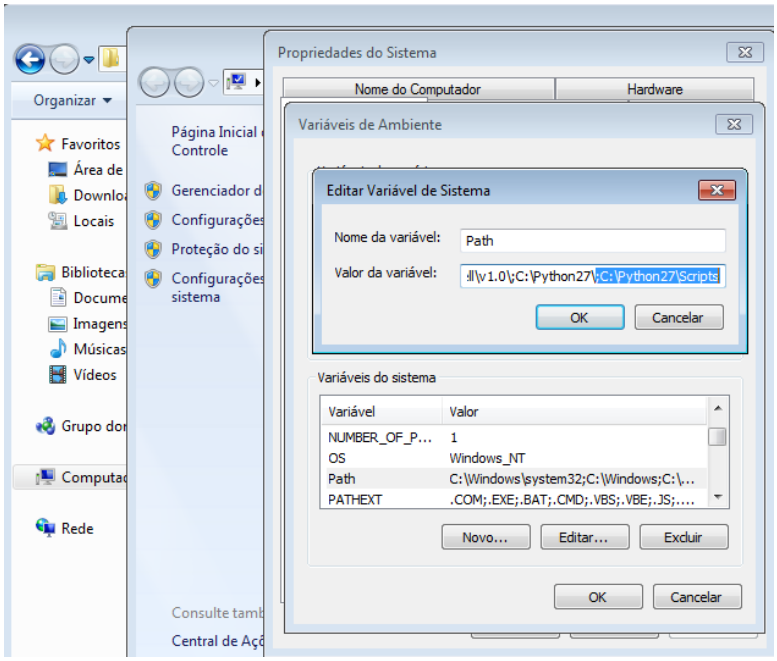



**Figura 3.01: Adicionado pip ao path**

O instalador atual do Python 2.7.9 para Windows já vem com pip instalado por padrão. Além disso, no processo de instalação é possível marcar opção para adição automática do Python ao path, evitando todo o trabalho manual.

Com o pip instalado deve ser executado o comando `pip install virtualenv` para instalar o virtualenv. Finalmente com o virtualenv instalado e o template de projeto extraído na "Área de

Trabalho" é possível criar o ambiente isolado para o projeto através da linha de comando. Para rodar o comando é necessário abrir o *prompt* como administrador. Para isso, clique no menu Iniciar, digite cmd. Mantenha pressionadas as teclas Crtl + Shift e então pressione enter:

**Criação de virtualenv**

```
C:\Users>cd renzovm
C:\Users\renzovm>cd Desktop
C:\Users\renzovm\Desktop>cd appengineepython
C:\Users\renzovm\Desktop\appengineepython>cd backend
C:\Users\renzovm\Desktop\appengineepython\backend>cd venv
C:\Users\renzovm\Desktop\appengineepython\backend\venv>venv.bat
```

Ao término da execução, o ambiente virtual será criado e as bibliotecas necessárias instaladas.

## Arquivo requirements.txt

As dependências de bibliotecas externas do projeto se encontram em dois arquivos: `requirements.txt` e `dev_requirements.txt`, presentes no diretório venv. No primeiro se encontram as dependências necessárias ao funcionamento do servidor:

**Arquivo requirements.txt**

```
tekton==4.0
gaebusiness==4.4.2
gaecookie==0.7
gaeforms==0.5
gaegraph==3.0.2
gaepermission==0.8
pytz==2014.4
Babel==1.3
python-slugify==0.0.7
```

Já o arquivo `dev_requirements` contém dependências necessárias apenas durante processo de desenvolvimento. Esse é o caso das bibliotecas Mock e Moomygae, utilizadas para facilitar a criação de testes automáticos:

**Arquivo dev_requirements.txt**

```
-r requirements.txt
mock==1.0.1
mommygae==1.1
```

Esses arquivos podem ser editados para conter dependências que se julguem necessárias durante o desenvolvimento.

## Links Simbólicos

Em um servidor comum o processo de instalação de dependências seria feito com os mesmos comandos. Contudo, não temos acesso ao Sistema Operacional do GAE. Sendo assim, é necessário informar ao sistema onde se encontram as bibliotecas, de forma que elas sejam copiadas durante o processo de *deploy*. Além das bibliotecas, também devem ser dispobilizados os códigos das aplicações presentes no diretórios `apps`.

Essas tarefas são realizadas pelo *script* de setup. Ele cria os links simbólicos `apps` e `lib` na pasta `appengine` apontado para os diretórios de interesse.

## Virtualenv e Pycharm

As ações realizadas até agora serviram apenas para instalar as bibliotecas no projeto. Contudo, o Pycharm precisa utilizar o ambiente criado para poder auxiliar no desenvolvimento, oferecendo opções de *auto complete* referente às bibliotecas e apps.

Para isso deve ser acessada a janela de configuração da IDE (ícone chave de roda). No *input* de pesquisa deve ser inserida a palavra *Interpreter* e escolhida a opção *Python Interpreters*. Deve ser pressionado o ícone "+" e escolhida a opção `Local...`, conforme figura 3.02:

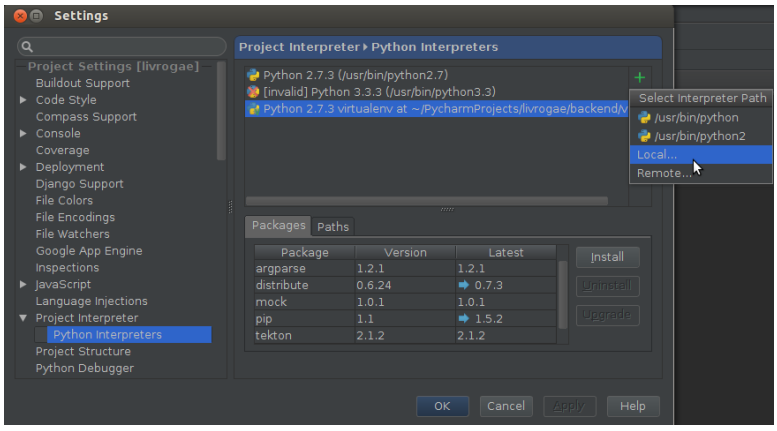



**Figura 3.02: Virtualenv e Pycharm**

Feito isso, deve ser selecionado o ambiente isolado localizado dentro do projeto. No Linux o arquivo a ser escolhido é `/backend/venv/bin/python`. Já no Windows ele se encontra em `\backends\venv\Scripts\python.exe`.

Indicado o virtualenv, deve ser editada a localização do servidor. Para isso deve ser acessado o menu "Edit Configurations..." conforme figura 3.03:

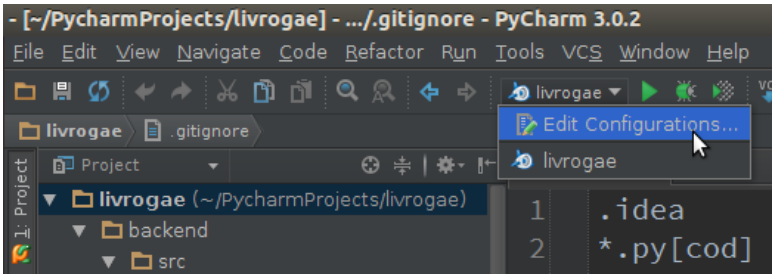



**Figura 3.03: Configuração de servidor no Pycharm**

Uma vez nessa janela, deve ser configurado o diretório `appengine` do projeto como "Working directory", já que ele contém o arquivo `app.yaml`. A figura 3.04 mostra a configuração final:

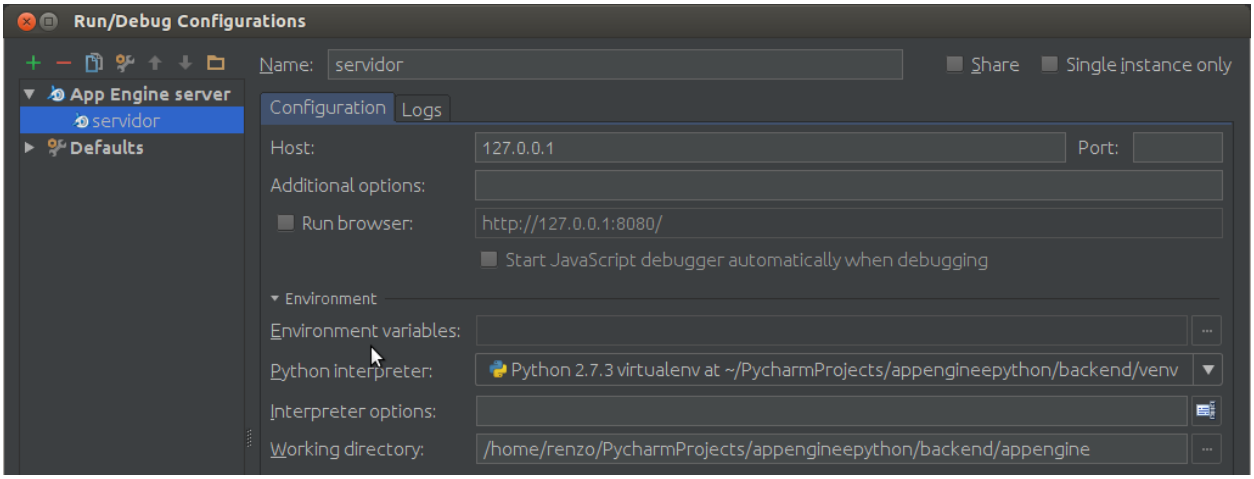



**Figura 3.04: Diretório do servidor**

Após toda essa configuração é possível executar o servidor local e verificar a página inicial do projeto no navegador, conforme figura 3.05:

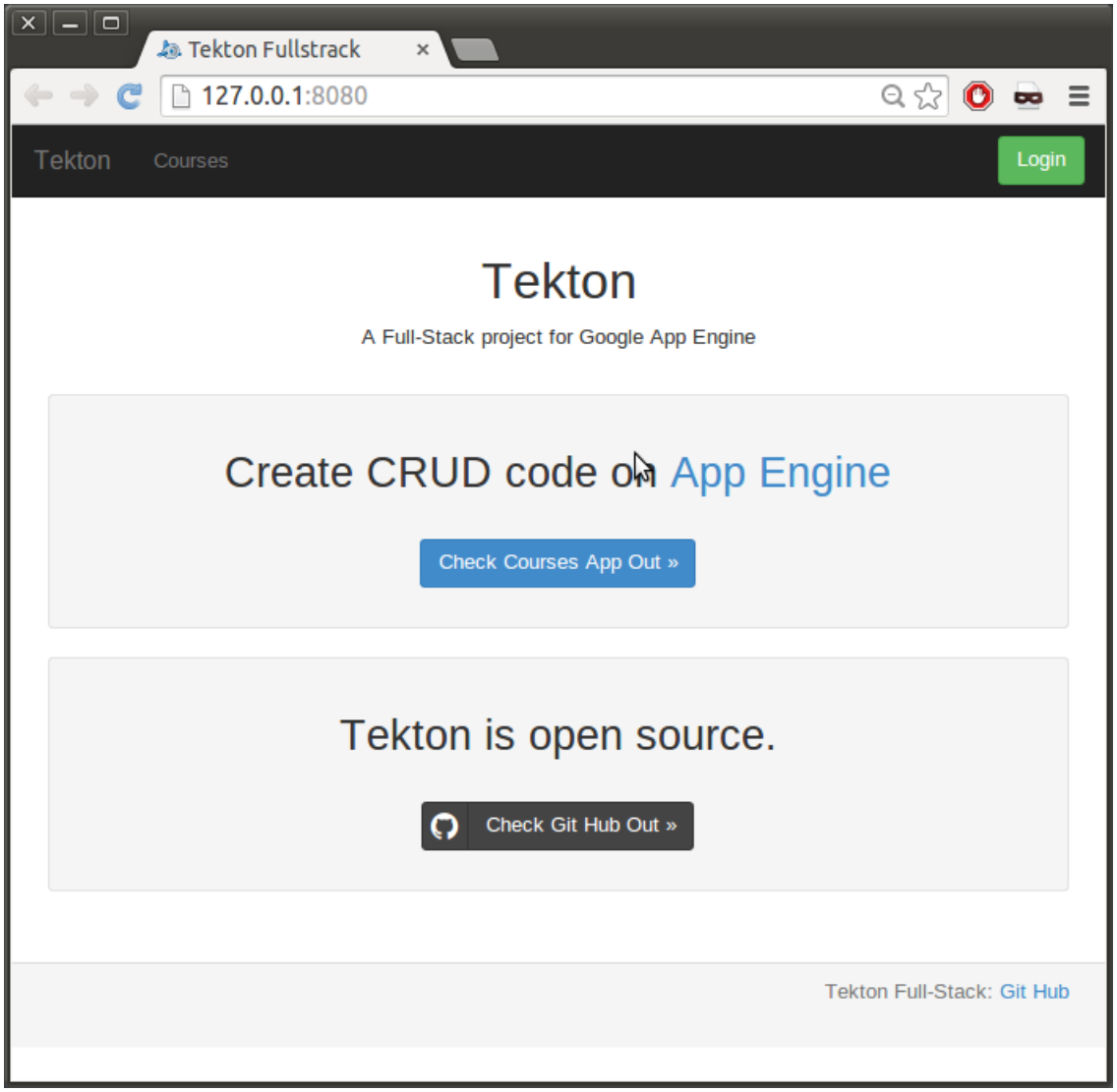



**Figura 3.05: Hello World Tekton**

Como último passo de configuração, é necessário então marcar pastas chave como raízes de código fonte. Isso é necessário para que a IDE consiga inspecionar seus conteúdos a fim de auxiliar no processo de desenvolvimento. A figura 3.06 mostra o menu a ser acessado quando se clica com botão direito do mouse sobre cada um dos diretórios:

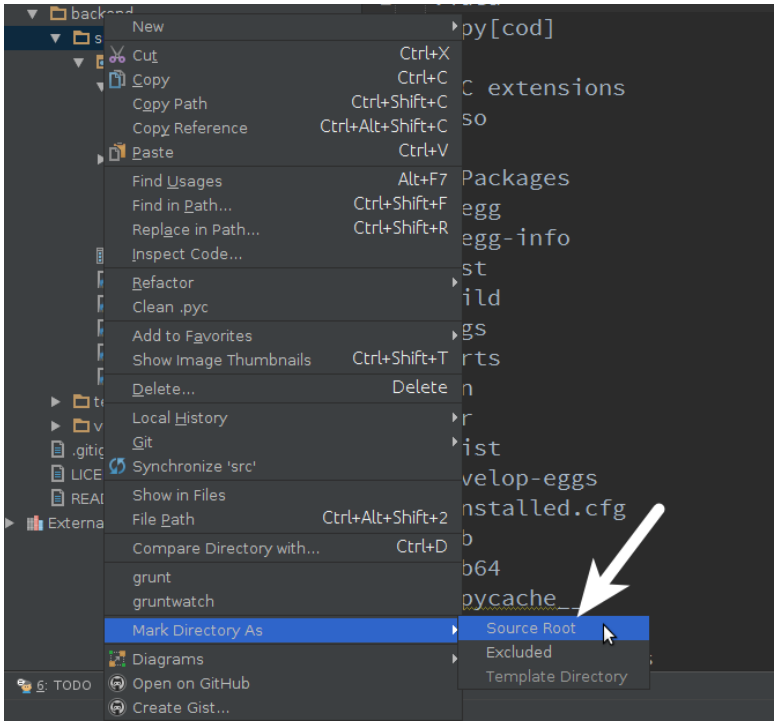



**Figura 3.06: Raiz de código fonte**

Os diretórios a serem marcados são:

1. apps;
2. test;
3. appengine.

Apesar de trabalhoso, a parte de configuração do projeto é feita somente uma vez. Com os ganhos de produtividade, o tempo gasto na configuração será recuperado rapidamente durante a fase desenvolvimento.

# Script convention.py

No projeto configurado o *framework* Tekton utiliza os objetos básicos do Webapp2. Sendo assim, é possível investigar o arquivo de configuração app.yaml para entender seu funcionamento:

**Seção handlers do arquivo app.yaml com Tekton configurado**

```
handlers:
- url: /
  *script*: convention.app
  secure: always

- url: /robots\.txt
  static_files: static/robots.txt
  upload: static/robots.txt

- url: /favicon\.ico
  static_files: static/img/favicon.ico
  upload: static/img/favicon.ico

- url: /static(.*)
  static_files: static\1
  upload: static.*

- url: /[^_].*
  *script*: convention.app
  secure: always

- url: /_ah/warmup
  *script*: routes.warmup.app
```

Da configuração é possível notar que as requisições serão tratadas, com apenas algumas exceções, pelo arquivo `convention.py`. O conteúdo desse *script* se encontra no código 3.01.

Nas linhas 2 e 3 do arquivo são adicionados os diretórios `lib` e `apps`, presentes em `appengine`, ao path. Essa é a razão de se ter criado os links simbólicos nesse endereço, na seção de *Setup Inicial*.

Nas linhas 11 e 12 são definidos os parâmetros de localização e fuso horário da aplicação. Mais detalhes sobre isso será visto na seção "Arquivo settings.py".

Nas demais linhas é definido um único `RequestHandler` que trata todas requisições. É importante ressaltar que tanto chamadas POST e GET são tratadas pelo mesmo método `make_convention`. Se for necessário atender outros métodos, como o PUT, é suficiente editar o arquivo copiando o método get e substituindo seu nome para também atender a respectiva chamada HTTP.

**Código 3.01: Script convention.py**

```
# Put lib on path, once Google App Engine does not allow doing it directly
sys.path.append(os.path.join(os.path.dirname(__file__), "lib"))
sys.path.append(os.path.join(os.path.dirname(__file__), "apps"))

import settings
from tekton.gae import middleware
import webapp2
from webapp2_extras import i18n

i18n.default_config['default_locale'] = settings.DEFAULT_LOCALE
i18n.default_config['default_timezone'] = settings.DEFAULT_TIMEZONE


class BaseHandler(webapp2.RequestHandler):
    def get(self):
        self.make_convention()

    def post(self):
        self.make_convention()

    def make_convention(self):
        middleware.execute(settings.MIDDLEWARE_LIST, self)


app = webapp2.WSGIApplication([("/.*", BaseHandler)], debug=False)
```

A ideia geral é a delegação do processamento de todas requisições pelo *script* `convention.py`, não mais sendo necessário mapear *handlers* manualmente. A maneira de se fazer isso será explicada na próxima seção.

# Roteamento via Tekton

O roteamento no *framework* Tekton é feito por convenção entre o conteúdo do pacote `routes`, presente no diretório `appengine`, e os *paths* da aplicação. Como exemplo inicial, ao se fazer o acesso à raiz "/" do projeto, a biblioteca irá procurar pelo módulo `home.py`, dentro do diretório `routes`, e executar sua função de nome `index`. O código 3.02 apresenta o código do arquivo:

**Código 3.02: home.py**

```
1  @login_not_required
2  @no_csrf
3  def index():
4      return TemplateResponse()
```

O resultado da execução do código foi visto no final da seção de Setup, na figura 3.05, onde aparecia a home do Tekton.

No caso do acesso ao *path* `/usuario`, a biblioteca irá procurar pelo *script* `usuario.py` e executar sua função `index`. Já o acesso a `/usuario/ola` acarretará na execução da função `ola` do *script* `usuario.py`. O arquivo se encontra sob o `pacote` routes:

```
routes/
├── home.py
├── usuario.py
```

O código 3.03 apresenta o conteúdo do *script*:

**Código 3.03: *script* usuario.py**

```
1  @login_not_required
2  @no_csrf
3  def index(_resp):
4      _resp.write('Página de Usuário')
5
6
7  @login_not_required
8  @no_csrf
9  def ola(_resp):
10     _resp.write('Olá Usuário')
```

Ao acessar o *link* `http://localhost:8080/usuario` é possível visualizar a mensagem "Página de Usuário" no navegador. De maneira análoga, ao se acessar `http://localhost:8080/usuario/ola` será visualizada a mensagem "Olá Usuário".

Seguindo esse esquema de convenção, podem ser criados pacotes, módulos e funções dentro de `routes`. Os *script*s serão sempre encontrados de acordo com o `path` acessado no navegador. Dessa maneira fica dispensada a configuração do roteamento que foi necessária no capítulo 2: Webapp2.

# Recebimento de Parâmetros

Além da convenção de rotas, o recebimento de parâmetros também é feito por convenção. Para receber o valor de um parâmetro chamado "nome" é necessário apenas declarar um argumento de mesmo nome. Como exemplo, a função `ola` foi alterada conforme código 3.04:

**Código 3.04: função ola com parâmetro nome**

```
1 def ola(_resp, nome):
2     _resp.write("Olá %s" % nome)
```

A figura 3.07 mostra o resultado da passagem de parâmetro via *query string*:

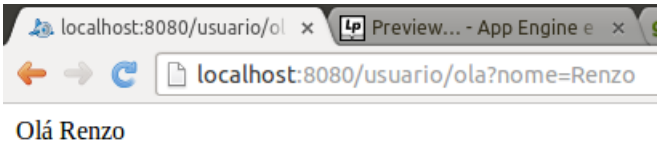



**Figura 3.07: execução de ola com parâmetro "nome" igual a "Renzo"**

Mais parâmetros podem ser recebidos acrescentando-se argumentos à função, conforme código 3.05:

**Código 3.09: função ola com parâmetros nome e sobrenome**

```
1 def ola(_resp, nome, sobrenome):
2     _resp.write("Olá %s %s" % (nome, sobrenome))
```

Acessando o endereço `http://localhost:8080/usuario/ola?nome=Renzo&sobrenome=Nuccitelli` é exibida no navegador a mensagem "Olá Renzo Nuccitelli".

Cabe ressaltar que a passagem de parâmetros pode ser feita de maneira *RESTful*. Ou seja, a mesma mensagem é obtida se os parâmetros forem passados através do endereço:

`http://localhost:8080/usuario/ola/Renzo/Nuccitelli`.

É possível ainda mesclar as duas formas. Como exemplo, o acesso ao endereço:

`http://localhost:8080/usuario/ola/Renzo?sobrenome=Nuccitelli`

A mensagem seria exatamente a mesma vista anteriormente.

# Configurações Globais e Internacionalização

O arquivo `settings.py` contém as configurações globais do sistema. Nele é possivel alterar endereço responsável por envio de emails pelo sistema, idioma, fuso-horário, entre outras. Como exemplo, o arquivo foi editado para português brasileiro e fuso de São Paulo:

**Arquvio setting.py**

```
APP_URL = 'https://tekton-fullstack.appspot.com'
SENDER_EMAIL = 'renzon@gmail.com'
DEFAULT_LOCALE = 'pt_BR'
DEFAULT_TIMEZONE = 'America/Sao_Paulo'
LOCALES = ['en_US', 'pt_BR']
TEMPLATE_404_ERROR = 'base/404.html'
TEMPLATE_400_ERROR = 'base/400.html'
```

Após alterar o idioma é necessário rodar o *script* para gerar o arquivo de traduções. O Tekton já possui embutida biblioteca de internacionalização, facilitando a construção de sites em múltiplas línguas. Maiores detalhes serão vistos em capítulos posteriores. Por ora, é suficiente rodar o *script* `build_*script*s/babel/i18n_extractor.py`. Gerados os arquivos, é possível verificar a home page traduzida:


**Figura 3.08: Home em português brasileiro**

# Injeção de Dependência

O *framework* Tekton provê um sistema simplificado de injeção de dependência por convenção de nomes. Ao declarar parâmetros com identificação especial, a biblioteca injeta objetos de interesse automaticamente. É o caso do parâmetro `_resp` constante no código 3.09, reproduzido abaixo por comodidade. Através dele se tem acesso ao objeto `Response`, visto em detalhes no capítulo 2: Webapp2.

**Código 3.09: função ola com parâmetros nome e sobrenome**

```
1 def ola(_resp, nome, sobrenome):
2     _resp.write("Olá %s %s" % (nome, sobrenome))
```

Se fosse necessário receber também o objeto `Request` como dependência, seria suficiente acrescentar um segundo parâmetro à função, conforme código código 3.10:

**Código 3.10: função ola com injeção de objetos Response e Request**

```
1 def ola(_resp, _req, nome, sobrenome):
2     _resp.write("Olá %s %s" % (nome, sobrenome))
3     # Imprimindo parametros de requisição http
4     _resp.write("Parametros: %s" % _req.arguments())
```

O *underscore* "_" na frente dos parâmetros injetados é apenas uma convenção. Ela foi adotada para diferenciar os argumentos que são injetados daqueles que são extraídos da requisição HTTP. Dessa maneira é possível perceber no código 3.10 que `_resp` e `_req` são dependências injetadas, enquanto `nome` e `sobrenome` são parâmetros recebidos via protocolo HTTP.

Os parâmetros injetados devem sempre ser os primeiros argumentos da função. Sendo assim, não seria possível trocar de posição os parâmetros `_resp` e `nome`

Da mesma forma que foram extraídas as dependências do *framework* Webapp2, a mesma ideia pode ser empregada para outros objetos ou funções. A vantagem dessa técnica é tornar o código testável. Isso ficará mais claro no capítulo de Testes Automatizados.

## Redirecionamento

Uma vez que o objeto `RequestHandler` é recebido como injeção de dependência, para fazer um redirecionamento é necessário apenas recebê-lo através do parâmetro `_handler` e utilizar seu método `redirect`. O código 3.11 mostra o método `redirecionar` que redireciona para o *path* respectivo à função `ola`:

**Código 3.11: Método Redirecionar com url como string, *script* usuario.py**

```
@login_not_required
@no_csrf
def redirecionar(_handler):
    url = r'/usuario/ola/Renzo/Nuccitelli'
    _handler.redirect(url)
```

O problema dessa abordagem é que o *path* é inserido como uma string. Se por alguma razão o nome da função `ola` for alterado, o redirecionamento levará a um link quebrado. Por essa razão o *Tekton* provê uma interface para calcular a url a partir de uma função.

A código 3.12 mostra o código alterado, fazendo uso do módulo `router`, presente no pacote `tekton`, para gerar a url baseada na função, que é seu primeiro parâmetro. Nessa abordagem, em caso de refatoração da função `ola`, o link do redirecionamento iria ser atualizado automaticamente para o endereço correto.

**Código 3.12: Método Redirecionar com url calculada por tekton.router.py**

```
def redirecionar(_handler):
    ~~url = r'/usuario/ola/Renzo/Nuccitelli'~~
    url = router.to_path(ola, 'Renzo', 'Nuccitelli')
    _handler.redirect(url)
```

Dessa maneira se encerra a explicação das funcionalidades básicas da biblioteca.

# Resumo

Nesse capítulo foi abordado o *framework* Tekton. Foi utilizado o virtualenv para sua instalação e o arquivo requirements.txt para resolução de dependências de outras bibliotecas.

Foram explicadas as várias convenções, visando evitar excessiva configuração:

- Roteamento via localização de módulos sob o pacote `routes`;
- Recebimento de parâmetro por convenção de nomes;
- Injeção de Dependências permitindo acesso a objetos do Weapp2;
- Utilização de configurações globais.

Por fim, foi visto como se utilizar o módulo `tekton.router.py` para calcular `paths`, permitindo a atualização automática de endereços no caso de refatoração de nomes de funções.

Com base nesse conhecimento serão construídos todos os exemplos constantes no restante desse livro.

# Questões

1. Para que serve o Virtualenv?
2. Qual a função do arquivo `convention.py`?
3. Por que é necessário incluir bibliotecas através de código no arquivo `convention.py`?
4. Como ficaria a declaração de uma função para tratar a execução de chamada no *path* `/usuario/salvar?nome=Renzo&idade=31`?
5. Como se diferenciam parâmetros recebidos por injeção de dependência dos recebidos via requisição HTTP?
6. Qual deve ser a posição de parâmetros recebidos via injeção de dependência?
7. Qual deve ser o parâmetro declarado quando for necessário fazer um redirecionamento?
8. Qual o *script* e função devem ser utilizados para se calcular *paths* com base em uma função?

# Respostas

1. O Virtualenv serve para se criar ambientes Python isolados para cada projeto.
2. O arquivo `convention.py` serve para fazer a ligação entre o Tekton e o Wepapp2. Ele é o arquivo onde se encontra o *handler* que delega todas requisições para funções que se encontram sob o pacote `routes`, através de convenção.
3. É necessário incluir bibliotecas através de código no arquivo `convention.py` porque diferente de servidores tradicionais, na plataforma GAE não se tem acesso ao Sistema Operacional para se poder instalar as bibliotecas utilizando virtualenv e pip.
4. A declaração da função seria `def salvar(nome,idade)` e deveria constar no *script* `usuario.py`.
5. Parâmetros recebidos via injeção de dependência possuem a convenção de conter o prefixo "_" antes de seus nomes. Parâmetros recebidos via requisição HTTP são escritos sem esse prefixo.
6. Parâmetros recebidos via injeção de dependência devem sempre ser os primeiros a serem declarados em uma função.
7. Quando for necessário fazer um redirecionamento deve ser declarada a dependência `_handler` para acessar a instância de `RequestHandler` que está tratando a requisição. Deve ser utilizado o método `redirect` desse objeto para se efetuar o redirecionamento.
8. Deve ser utilizada a função `to_path` do módulo `tekton.router` para se calcular o *path* respectivo a uma função, que deve ser passada como parâmetro.