Python for Google App Engine

Google App Engine is a key component of the Google Cloud Platform and one of the most comprehensive cloud services. Together with Python, the first programming language ever supported by the platform, Google App Engine favors rapid development, significantly contributing to the success of your projects.

This book will guide you through the Google Cloud Platform starting from the basics all the way through successfully designing, building, and running modern and scalable web applications in Python with Google App Engine. From authentication to data storage, and from asynchronous jobs to server push techniques, you will learn how to use services such as datastore, cloud storage, Cloud SQL, task queues and channels in a Python application based on web frameworks such as Webapp2 and Django. As the topics in this book are covered in a step-by-step fashion, you will get to grips with the information quickly and easily.

Who this book is written for
If you are a Python developer, whether you have experience in web applications development or not, and want to rapidly deploy a scalable backend service or a modern web application on Google App Engine, then this book is for you.

What you will learn from this book
- Persist and manage data in the cloud datastore with the NDB Python API
- Run asynchronous tasks with task queue and Cron
- Set up, use, and manage a MySQL server instance on Cloud SQL
- Employ channels to develop real-time applications with push notifications from the server
- Write a complete Django application using Cloud SQL as the data backend
- Use cloud endpoints to rapidly provide REST APIs for your mobile clients

Massimiliano Pippi

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Getting Started"
- A synopsis of the book’s content
- More information on Python for Google App Engine

About the Author

Massimiliano Pippi has been a software developer for over 10 years, more than half of which he spent working with scientific visualization and backend software for a private company, using C++ and Qt technologies. He started using Python in 2008 and currently works at Evonove, a small company where he has been leading a number of Python software projects, most of which are based on the Django web framework. He is also an open source advocate and active contributor, documentation fanatic, and speaker at conferences. He writes about Python and other development-related topics at http://dev.pippi.im.
Python for Google App Engine

In April 2008, 10,000 developers from all around the world were lucky enough to get an account to access the preview release of Google App Engine, which is a tool designed to let users run their web applications on the same infrastructure Google uses for its own services. Announced during Google's Campfire One event, App Engine was described as something easy to use, easy to scale and free to get started; three design goals that perfectly matched the requirements of a typical tech start-up trying to reduce the time to market.

While other big companies at that time were already offering to lease part of their own infrastructure, selling reliability and scalability in an affordable, pay-per-use fashion, Google set App Engine one step ahead by providing developers with application-building blocks instead of simple access to hardware; it is a hosting model followed by many others later on. The goal of this model is to let developers focus on the code and forget about failing machines, network issues, scalability problems, and performance tuning; the choice of Python as the first programming language supported by App Engine was a natural choice for a tool whose aim is to make writing and running web applications easier.

During the Google I/O event in 2012, Google announced that several other building blocks from its own infrastructure would be made available under the name of Google Cloud Platform, first as a partner program and then as a general availability product. Currently, App Engine is not only a notable member of the Cloud Platform family but also a mature and well-maintained platform, widely adopted and with a huge list of customers' success stories.

This book will teach you how to write and run web applications in Python with App Engine, getting the most out of Google Cloud Platform. Starting with a simple application, you will add more and more features to it, each time with the help of the components and services provided by Google's infrastructure.

What This Book Covers

Chapter 1, Getting Started, will help you get your hands dirty with a very simple functional Python application running on a production server. The chapter begins with making a survey of Google's cloud infrastructure, showing where App Engine is placed and how it compares to other well-known cloud services. It then walks readers through downloading and installing the runtime for Linux, Windows, and OS X, coding a Hello, World! application and deploying it on App Engine. The last part introduces administration consoles both for the development and production servers.
Chapter 2, *A More Complex Application*, teaches you how to implement a complex web application running on App Engine. It begins with an introduction to the bundled webapp2 framework and possible alternatives; then, you will get in touch with user authentication and form handling and then an introduction to Google's Datastore nonrelational database. The last part shows you how to make HTML pages through templates rendering and how to serve all the static files needed to style the page.

Chapter 3, *Storing and Processing Users' Data*, will show you how to add more functionalities to the app from the previous chapter. The chapter begins by showing you how to let users upload files using Google Cloud Storage and how to manipulate such files when they contain image data with the Image API. It then introduces you to the task queues used to execute long jobs (such as image manipulation) outside the request process and how to schedule batches of such jobs. The last part shows you how to send and receive e-mails through the Mail API.

Chapter 4, *Improving Application Performance*, begins by showing how to improve application performance using advanced features of Datastore. It then shows you how to use the cache provided by App Engine and how to break the application into smaller services using Modules.

Chapter 5, *Storing Data in Google Cloud SQL*, is dedicated to the Google Cloud SQL service. It shows you how to create and manage a database instance and how to connect and perform queries. It then demonstrates how an App Engine application can save and retrieve data and how to use a local MySQL installation during development.

Chapter 6, *Using Channels to Implement a Real-time Application*, shows you how to make our application real time, in other words, how to update what clients see without reloading the page in the browser. The first part shows how the Channel API works, what happens when a client connects, and what roundtrip of a message is from the server to the client. Then, it shows you how to add a real-time feature to our application from previous chapters.

Chapter 7, *Building an Application with Django*, teaches you how to build an App Engine application using the Django web framework instead of webapp2. The first part shows you how to configure the local environment for development, and then the application from previous chapters is rewritten using some of the features provided by Django. The last part shows you how to deploy the application on a production server.

Chapter 8, *Exposing a REST API with Google Cloud Endpoints*, shows you how to rewrite part of our application to expose data through a REST API. The first part explores all the operations needed to set up and configure a project and how to implement a couple of endpoints for our API. The last part shows explores how to add OAuth protection to the API endpoints.
Any software available over the Internet, usually accessed with a web browser, can be addressed as a web application. Social networks, e-commerce sites, e-mail clients, online games are just a few examples of a trend known as web 2.0, which was started in the late 1990s and emerged in the past few years. Today, if we want to provide a service for multiple clients and multiple users, we will likely end in with writing a web application.

Web applications come with an endless list of benefits from a developer's point of view but there is one major drawback to face every time we want to make our software available to other users: we need a remote server connected to the Internet to host the application. This server must be constantly available and respond to clients in a reasonable amount of time, irrespective of the number of clients, or the application won't be usable.

A noteworthy solution to the hosting problem is cloud computing, which is a rather generic term that usually refers to the opportunity to run applications and services on someone else's infrastructure at a reasonable cost and in a way that is simple and quick for the needed resources to be provisioned and released.

In this first chapter we will define in detail the term cloud computing and then introduce the model provided by Google, focusing on the elements that are important to us, as developers, and use them to run our first application using the Google Cloud Platform and Google App Engine.

In this chapter we will cover the following topics:

- A detailed introduction to Google Cloud Platform and Google App Engine
- Setting up an App Engine code environment
- Writing a simple application
- Loading and running the application on a remote server
- Using the administration console
The cloud computing stack – SaaS, PaaS, and IaaS

We can choose to outsource our applications and the hardware they run on, still being responsible for the whole software stack, including the operating system; or, we can simply use existing applications available from another vendor.

We can represent cloud computing as a stack of three different categories: **Software as a Service (SaaS)**, **Platform as a Service (PaaS)**, and **Infrastructure as a Service (IaaS)** as follows:

```
   SaaS
  /    /
PaaS   IaaS
```

In the first case, the cloud computing model is defined as IaaS and we basically outsource hardware and every inherent service such as power supply, cooling, networking, and storage systems. We decide how to allocate resources, how many web applications, or database servers we need, whether or not we need to use a load balancer, how to manage backups and so on; the installation, monitoring, and maintenance are under our responsibilities. A notable example of IaaS services are EC2 from Amazon and Rackspace Cloud Hosting.

In the second case, the cloud computing model is defined as SaaS and is the opposite of IaaS since we simply use a turnkey software provided by a third-party vendor, who has no technical knowledge of the infrastructure it runs on; the vendor is responsible for the reliability and security of the product. Notable examples of SaaS are Gmail from Google and Salesforce.

Between IaaS and SaaS we find the PaaS model, which seems to be the most interesting solution from a developer’s point of view. A PaaS system provides a platform with which we can build and run our application without worrying about the underlying levels, both hardware and software.
Google Cloud Platform

Google Cloud Platform is designed to offer developers tools and services needed to build and run web applications on Google's reliable and highly scalable infrastructure. The platform consists of several cloud computing products that can be composed and used according to our needs, so it's important to know what these building blocks can do for us, as developers, and how they do so.

As we can learn from the main documentation page at https://cloud.google.com, Google classifies Google Cloud Platform's components into four groups: Hosting + Compute, Storage, Big Data, and services.

Hosting + Compute

There are two options if we want to host an application on Google Cloud Platform:

- **Google App Engine**: This is Google's PaaS and it will be covered in detail later in this chapter.
- **Google Compute Engine**: This is Google's IaaS and lets users run virtual machines on Google's infrastructure with a variety of hardware and software configurations.

Storage

Google Cloud Platform provides several options to store and access users' data:

- **Google Cloud Storage**: This is a highly available and scalable file storage service with versioning and caching. We will learn how to use Cloud Storage in Chapter 3, Storing and Processing User's Data.

- **Google Cloud SQL**: This is a fully managed MySQL relational database; replication, security and availability are Google's responsibilities. Chapter 5, Storing Data in Google Cloud SQL, is entirely dedicated to this service.

- **Google Cloud Datastore**: This is a managed schemaless database that stores nonrelational data objects called entities; it scales automatically, supports transactions, and can be queried with SQL-like syntax. We will start using it in Chapter 2, A More Complex Application, and learn how to get the most out of it in Chapter 4, Improving Application Performance.
BigQuery

BigQuery is a tool provided by Google Cloud Platform that allows to perform queries using an SQL-like syntax against a huge amount of data in a matter of seconds. Before it can be analyzed, data must be streamed into BigQuery through its API or uploaded to Google Cloud Storage.

Services

Instead of writing code from scratch, we can easily add functionalities to our applications using some of Google's services through APIs that are very well integrated within Google Cloud Platform:

- **The Translate API**: This can translate text between dozens of languages programmatically, from within our applications.
- **The Prediction API**: This predicts future trends using Google's machine learning algorithms and can be used from within our applications or through a **Representational State Transfer (REST)** API. REST is a stateless architecture style that describes how a system can communicate with another through a network; we will delve into more details on REST in *Chapter 8, Exposing a REST API with Google Cloud Endpoints*.
- **Google Cloud Endpoints**: Using this tool, it's easy to create applications that expose REST services, providing also **Denial-of-Service (DoS)** protection and **OAuth2** authentication. We will learn how to use them in *Chapter 8, Exposing a REST API with Google Cloud Endpoints*.
- **Google Cloud DNS**: This is global **Domain Name System (DNS)** service that runs on Google's infrastructure and provides high volume serving that is programmable from within our applications.
- **Google Cloud Pub/Sub**: This is middleware that provides many-to-many, asynchronous messaging between services that either run on Google Cloud Platform or externally.

All the tools and services provided by Google Cloud Platform are billed with a pay-per-use model so that applications can scale up or down as needed and we only pay for resources we actually use. A handy calculator is provided to have a precise idea of the costs depending on the services and resources we think we will need. Google Cloud Platform offers a certain amount of resources we can use without paying anything; usually, these free quotas are well suited to host web applications with low traffic at no cost.
What Google App Engine does

As mentioned earlier, App Engine is a PaaS, which means that we have the benefits of SaaS products but with an augmented flexibility as we have complete control over the code. We also have the benefits of an IaaS solution but without the hassle of maintaining and configuring the software environment needed to run applications on a raw hardware system.

Developers are the favored users of a PaaS product such as App Engine because the platform helps them in two ways: it provides an easy way to deploy, scale, tune, and monitor web applications without the need for a system administrator and it offers a set of tools and services that speed up the software development process. Let's explore these two aspects in detail.

The runtime environment

App Engine runs on computing units that are completely managed called instances. We can (and should) ignore which operating system is running on an instance because we interact solely with the runtime environment, which is an abstraction of the operating system that provides resource allocation, computation management, request handling, scaling, and load balancing.

Developers can choose among four different programming languages to write applications on App Engine: Python, Java, Hypertext Preprocessor (PHP), and Go but we will focus on the Python environment.

Every time a client contacts an application that runs on App Engine, a component of the runtime environment called scheduler selects an instance that can provide a fast response, initializes it with application data if needed, and executes the application with a Python interpreter in a safe, sandboxed environment. The application receives the HTTP request, performs its work, and sends an HTTP response back to the environment. Communication between the runtime environment and the application is performed using the Web Server Gateway Interface (WSGI) protocol; this means that developers can use any WSGI-compatible web framework in their application.

WSGI is a specification that describes how a web server communicates with web applications written in Python. It was originally described in PEP-0333 and later updated in PEP-3333, mainly to improve usability under the Python 3.0 release.
The runtime environment is sandboxed to improve security and provide isolation between applications running on the same instance. The interpreter can execute any Python code, import other modules, and access the standard library, provided that it doesn't violate sandbox restrictions. In particular, the interpreter will raise an exception whenever it tries to write to the filesystem, perform network connections, or import extension modules written in the C language. Another isolation mechanism we must be aware of that is provided by sandboxing, prevents an application from overusing an instance by raising an exception whenever the entire request/response cycle lasts more than 60 seconds.

Thanks to sandboxing, the runtime can decide at any given time whether to run an application on one instance or many instances, with requests being spread across all of them depending on the traffic. This capability, together with load balancing and scheduler settings is what makes App Engine really scalable.

Users can easily tune an application's performance by increasing its responsiveness or optimizing costs with a simple and interactive administrative console. We can specify instance performance in terms of memory and CPU limits, the number of idle instances always ready to satisfy a request, and the number of instances dynamically started when the traffic increases. We can also specify the maximum amount of time in milliseconds we tolerate for a pending request and let App Engine adjust the settings automatically.

The services
At first sight, restrictions imposed by the runtime environment might seem too restrictive. In the end, how can developers make something useful without being able to write data on disk, receive incoming network connections, fetch resources from external web applications, or start utility services such as a cache? This is why App Engine provides a set of higher-level APIs/services that can be used by developers to store and cache data or communicate over the Internet.

Some of these services are provided by the Google Cloud Platform as standalone products and are smoothly integrated into App Engine, while some others are only available from within the runtime environment.
Chapter 1

The list of available services changes quite often as Google releases new APIs and tools; the following is a subset of tools we will use later in the book in addition to the Datastore, Google Cloud Endpoints, Google Cloud SQL, and Google Cloud Storage services we introduced earlier:

- **Channel**: This API allows applications to create persistent connections with the clients and push data through such connections in real time instead of using polling strategies. Clients must use some JavaScript code to interact with the server. We will learn how to use Channels in *Chapter 6, Using Channels to Implement a Real-time Application.*

- **Datastore backup/restore**: At any given time, it’s possible to perform a backup of the entities contained in the Datastore or restore them from a previous backup; management operations are very easy as they can be performed interactively from the administrative console. We will see backup and restore procedures in detail in *Chapter 4, Improving Application Performance.*

- **Images**: This API lets developers access and manipulate image data provided by the application or loaded from Google Cloud Storage. We can get information about the format, size, and colors and perform operations such as resizing, rotating, and cropping and we can convert images between different formats and apply some basic filters provided by the API. We will use some of the features provided by the Images API in *Chapter 3, Storing and Processing Users’ Data.*

- **Mail**: This service allows applications to send e-mails on behalf of the administrators or users who are logged in with a Google Account and to receive e-mail messages sent to certain addresses and routed to the application. We will use both these features provided by the service in *Chapter 3, Storing and Processing Users’ Data.*

- **Memcache**: This is a general-purpose, distributed memory caching system that can be used to dramatically improve application performance, serving frequently accessed data way faster than accessing a database or an API. We will see how to use Memcache in *Chapter 4, Improving Application Performance.*

- **Modules**: These are used to split applications into logical components that can communicate and share their state with each other. They can be extremely useful as each of them can have different versions and performance and scaling settings, which provide developers with a great level of flexibility when tuning an application. We will see how to use Modules in *Chapter 4, Improving Application Performance.*
Getting Started

• **Scheduled tasks**: This is how App Engine implements the cron jobs. Developers can schedule a job to be executed at a defined date or at regular intervals. Schedules are defined in an English-like format: for example, `every Friday 20:00` is a valid schedule we can use to send weekly reports to our users. We will see how to use scheduled tasks in *Chapter 3, Storing and Processing Users' Data*.

• **Task Queue**: As mentioned earlier, the entire request/response cycle of an application running on App Engine must last at most 60 seconds, making it impossible to perform long operations. This is why the Task Queue API exists—it can perform work outside the user request so that long operations can be executed later in background with 10 minutes to finish. We will see how to use a task queue in *Chapter 3, Storing and Processing Users' Data*.

• **URL Fetch**: As we already know, the runtime environment prevents our application from performing any kind of network connection but accessing external resources through HTTP requests is a common requirement for a web application. This limitation can be overcome using the URL Fetch API to issue HTTP or HTTPS requests and retrieve a response in a scalable and efficient manner.

• **Users**: We can authenticate users within our applications using Google Accounts, accounts in a Google Apps domain, or through OpenID identifiers. Using the Users API our application can determine whether a user is logged in and redirect them to the login page or access their e-mail otherwise. Using this API, developers can delegate to Google or to the OpenID provider the responsibility of creating accounts and verifying the user's data.

For more information on the tools and services provided by Google that we can use from within the App Engine environment, refer to [https://developers.google.com/appengine/features/](https://developers.google.com/appengine/features/).

**Making our first Python application**

We have now have an idea of the features Google Cloud Platform can provide us with and we are ready to put App Engine in action, but before we can start writing some code, we need to set up our workstation.
Download and installation

To get started, we need to install the Google App Engine SDK for Python for the platform of our choice. The SDK contains all the libraries needed to develop an application and a set of tools to run and test the application in the local environment and deploy it in the production servers. On some platforms, administrative tasks can be performed through a GUI, the Google App Engine Launcher, on other platforms we can use a comprehensive set of command line tools. We will see Google App Engine Launcher in detail later in this chapter.

Before installing the SDK, we have to check whether a working installation of Python 2.7 (version 2.7.8 is the latest at the time of writing this book) is available on our system; we need this specific version of Python because, with 2.5 deprecated now, it is the only version supported by the App Engine platform. If we are using Linux or Mac OS X, we can check the Python version from the terminal that issues the command (notice the capital letter v):

```
python -V
```

The output should look like this:

```
Python 2.7.8
```

If we are on Windows, we can just ensure the right version of Python is listed in the Programs section within the Control Panel.

The official App Engine download page contains links for all the available SDKs. The following link points directly to the Python version: https://developers.google.com/appengine/downloads#Google_App_Engine_SDK_for_Python.

We have to choose the right package for our platform, download the installer, and proceed with the installation.

Installing on Windows

To install the SDK on Windows we have to download the .msi file from the App Engine download page, double-click it to launch the installation wizard, and follow the instructions on the screen. Once the install is complete, a shortcut to Google App Engine Launcher will be placed on the desktop as well as an item within the Start menu. The Windows version of the SDK does not provide any command-line tool, so we will always use Launcher to manage our applications.
Installing on Mac OS X

To install the SDK on Mac OS X, we have to download the .dmg file from the App Engine download page, double-click it to open the disk image, and drag the App Engine icon into the Applications folder. It is convenient to keep a shortcut to Launcher in our Dock; to do so, we just have to just drag the App Engine icon again from the Applications folder to the dock. The command-line tools will also be installed and during the first execution of Launcher, a pop-up dialog will prompt us as to whether we want to create the symlinks needed to make the tools available system-wide, so they can be executed from any terminal window without further configuration.

Installing on Linux

To install the SDK on Linux and more generally on POSIX-compliant systems, we have to download the .zip file from the App Engine download page and extract its contents in a directory of our choice. The archive contains a folder named google_appengine that contains the runtime and the command-line tools, and we have to add it to our shell's $PATH environment variable to make the tools available from within any terminal. The Linux version of the SDK does not include Launcher.

App Engine Launcher

The Windows and OS X versions of the SDK ships with a graphical user interface tool called Launcher that we can use to perform administrative tasks such as creating and managing multiple applications.

Launcher is a very handy tool but bear in mind that while every single task we can accomplish through Launcher can be performed by command-line tools as well, the contrary isn't true. There are tasks that can be performed only from the command line using the proper tools as we will see later in the book.
The following screenshot shows the Launcher window in OS X:

![Launcher window in OS X](image)

We can see the Launcher in Windows in the following screenshot:

![Launcher window in Windows](image)
Before we start using the Launcher it’s important to check whether it is using the right Python version. This is very important if we have more than one Python installation in our system. To check the Python version used by Launcher and to change it, we can open the Preferences... dialog by clicking the appropriate menu depending on our platform and set the Python path value. In the same dialog we can specify which text editor Launcher will open by default when we need to edit application files.

To create a new application we can click New Application in the File menu or click the button with a plus sign icon in the bottom-left corner of the Launcher window. Launcher will prompt for the application name and the path to the folder that will contain all the project files; once created, the application will be listed in the main window of Launcher.

We can start the local development server by clicking the Run button on the Launcher toolbar or clicking Run in the Control menu. Once the server is started, we can stop it by clicking on the Stop button or the Stop entry in the Control menu. Clicking the Browse button or the Browse entry in the Control menu opens the default browser at the home page of the selected application. To browse the logs produced by the development server, we can open the Log Console window by clicking the Logs button on the toolbar or the Logs entry in the Control menu. The SDK Console button on the toolbar and the SDK Console action on the Control menu will open the default browser at the URL that serves the Developer Console, a built-in application to interact with the local development server, which we will explore in detail later in this chapter.

The Edit button will open the configuration file for the selected application in an external text editor, maybe the one we specified in the Preferences... dialog; the same happens when we click the Open in External Editor action in the Edit menu.

To deploy and upload the selected application to App Engine we can click the Deploy button on the toolbar or click the Deploy action in the Control menu. The Dashboard button on the toolbar and the Dashboard action in the Control menu will open the default browser at the URL of App Engine Administrative Console.

Using Launcher we can set additional flags for the local development server and customize some parameters such as the TCP port number to which listens. To do so we have to click the Application Settings... entry in the Edit menu and make the desired adjustments in the settings dialog.

Launcher can also handle existing applications created from scratch through the command line or checked out from an external repository. To add an existing application to the Launcher, we can click the Add Existing Application... entry in the File menu and specify the application path.
Creating the application

The first step to create an application is pick a name for it. According to the tradition we're going to write an application that will print "Hello, World!" so we can choose helloworld as the application name. We already know how to create an application from Launcher, the alternative is to do it manually from the command line.

At the simplest, a working Python application consists of a folder called application root that contains an app.yaml configuration file and a Python module with the code needed to handle HTTP requests. When we create an application within Launcher, it takes care of generating those files and the root folder for us, but let's see how can we can accomplish the same result from the command line.

The app.yaml configuration file

When we start creating the root folder, it doesn't matter how we name it but to be consistent with Launcher we can use the application's name:

```
mkdir helloworld && cd helloworld
```

We then create an app.yaml file that contains the following YAML code:

```yaml
application: helloworld
version: 1
runtime: python27
api_version: 1
threadsafe: yes

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

libraries:
  - name: webapp2
    version: "2.5.2"
```

YAML (a recursive acronym for YAML Ain't Markup Language) is a human-readable serialization format that is suitable for configuration files that have to be accessed and manipulated both from users and programatically.
The first section of the previous code defines some setup parameters for the application:

- **The application parameter**: This is the application name; later in the chapter, we'll see how important it is.
- **The version parameter**: This is a string that specifies the version of the application. App Engine retains a copy of each version deployed and we can run them selectively, a very useful feature for testing an application before making it public.
- **The runtime parameter**: At the time of writing this book, Python 27 is the only runtime available for newly created applications as Python 25 was deprecated.
- **The api_version parameter**: This is the version of the API for the current runtime environment. At the time writing this, 1 is the only API version available for the Python 27 runtime.
- **The threadsafe parameter**: This specifies whether our application can handle requests concurrently in separate threads or not.

The next section of the app.yaml file lists the URLs we want to match in the form of a regular expression; the script property specifies the handler for each URL. A handler is a procedure App Engine invokes to provide a response when an application receives a request. There are two types of handlers:

- **The script handlers**: These handlers run the Python code provided by the application.
- **The static file handlers**: These handlers return the content of a static resource such as an image or a file that contain the JavaScript code.

In this case, we are using a script handler, a Python callable addressed with a dot notation import string: App Engine will match any URL and invoke the app object contained in the main module.

The final section lists the name and version of third-party modules provided by App Engine we want to use from our application, and in this case we only need the latest version of the webapp2 web framework. We might wonder why we need something complex such as a web framework to simply print a "Hello, World!" message, but as we already know, our handler must implement a WSGI-compliant interface and this is exactly one of the features provided by webapp2. We will see how to use it in the next section.
The main.py application script

Now that the application is configured, we need to provide logic, so we create a file named main.py in the application root folder that will contain the following:

```python
import webapp2

class MainHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('Hello world!')

app = webapp2.WSGIApplication([
    ('/', MainHandler)
], debug=True)
```

In the first line of the previous code we import the `webapp2` package into our code, and then we proceed to define a class named `MainHandler` that is derived from the `RequestHandler` class provided by the framework. The base class implements a behavior that makes it very easy to implement a handler for HTTP requests; all we have to do is to define a method named after the HTTP action we want to handle. In this case, we implement the `get()` method that will be automatically invoked whenever the application receives a request of the type `GET`. The `RequestHandler` class also provides a `self.response` property we can use to access the response object that will be returned to the application server. This property is a file-like object that supports a `write()` method we can use to add content to the body of the HTTP response; in this case we write a string inside the response body with the default content type `text/html` so that it will be shown inside the browser.

Right after the `MainHandler` class definition we create the `app` object, which is an instance of the `WSGIApplication` class provided by `webapp2` that implements the WSGI-compliant callable entry point we specified in `app.yaml` with the import string `main.app`. We pass two parameters to the class constructor, a list of URL patterns, and a Boolean flag stating whether the application should run in debug mode or not. URL patterns are tuples that contain two elements: a regular expression that matches requested URLs and a class object derived from `webapp2.RequestHandler` class that will be instantiated to handle requests. URL patterns are processed one by one in the order they are in the list until one matches and the corresponding handler is called.

As we may notice, URL mappings take place twice—firstly in the `app.yaml` file, where a URL is routed to a WSGI compatible application in our code and then in the `WSGIApplication` class instance, where an URL is routed to a request handler object. We can freely choose how to use these mappings, that is either route all URLs in the `app.yaml` file to a single `webapp2` application where they are dispatched to handlers or to different URLs to different, smaller `webapp2` applications.
Running the development server

The App Engine SDK provides an extremely useful tool called development server that runs on our local system emulating the runtime environment we will find in production. This way, we can test our applications locally as we write them. We already know how to start the development server from Launcher. To launch it from the command line instead, we run the dev_appserver.py command tool passing the root folder of the application we want to execute as an argument. For example, if we're already inside the root folder of our helloworld application, to start the server, we can run this command:

dev_appserver.py .

The development server will print some status information on the shell and will then start listen at the local host to the default TCP ports 8000 and 8080, serving the admin console and the application respectively.

While the server is running, we can open a browser, point it at http://localhost:8080 and see our first web application serving content.

The following screenshot shows the output:
If we are using Launcher, we can simply press the **Browse** button and the browser will be opened automatically at the right URL.

The development server automatically restarts application instances whenever it detects that some content on the application root folder has changed. For example, while the server is running we can try to change the Python code that alters the string we write in the response body:

```python
import webapp2

class MainHandler(webapp2.RequestHandler):
    def get(self):
        self.response.write('<H1>Hello world!</H1>')
        self.response.write('<p>I’m using App Engine!</p>

app = webapp2.WSGIApplication([="/", MainHandler], debug=True)
```

After saving the file, we can refresh the browser and immediately see the changes without reloading the server, as shown in the following screenshot:

![Browser Screenshot]

We can now move our application to a production server on App Engine and make it available through the Internet.
Uploading the application to App Engine

Every application running on App Engine is uniquely identified by its name within the Google Cloud Platform. That is why sometimes we find parts of the documentation and tools referring to that as application ID. When working on a local system, we can safely pick any name we want for an application as the local server does not enforce any control on the application ID; but, if we want to deploy an application in production, the application ID must be validated and registered through App Engine Admin Console.

Admin Console can be accessed at https://appengine.google.com/ and log in with a valid Google user account or a Google apps account for custom domains. If we are using Application Launcher, clicking the Dashboard button will open the browser at the right address for us. Once logged in, we can click the Create Application button to access the application creation page. We have to provide an application ID (the console will tell us whether it is valid and available) and a title for the application and we're done. For now, we can accept the default values for the remaining options; clicking on Create Application again will finally register the application's ID for us.

Now we have to change the dummy application ID we provided for our application with the one registered on App Engine. Open the app.yaml configuration file and change the application property accordingly:

```yaml
application: the_registered_application_ID
version: 1
runtime: python27
api_version: 1
threadsafe: yes

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

libraries:
- name: webapp2
  version: "2.5.2"
```
We are now ready to deploy the application on App Engine. If we are using Application Launcher, all we have to do is click on the **Deploy** button in the toolbar. Launcher will ask for our Google credentials and then the log window will open showing the deployment status. If everything went fine the last line shown should be something like this:

*** appcfg.py has finished with exit code 0 ***

Deploying from the command line is just as easy; from the application root directory, we issue the command:

```
appcfg.py update .
```

We will be prompted for our Google account credentials, and then the deployment will proceed automatically.

Every App Engine application running in production can be accessed via http://the_registered_application_ID.appspot.com/, so we can tell whether the application is actually working by accessing this URL from a browser and checking whether the output is the same as that produced by the local development server.

Google App Engine allow us to serve content over **HTTPS (HTTP Secure)** connections on top of the **Secure Sockets Layer (SSL)** protocol, which means that data transferred from and to the server is encrypted. When using the appspot.com domain, this option is free of charge. To enable secure connections between clients and the App Engine server, all we have to do is add the `secure` option to the URLs listed in the app.yaml file:

```
handlers:
- url: .*
  script: main.app
  secure: always
```

On the local development server we will still use regular HTTP connections, but in production we will access https://the_registered_application_ID.appspot.com/ in a secure manner over HTTPS connections.

If we want to access the application over HTTPS through a custom domain instead, such as example.com, we have to configure App Engine so that the platform can use our certificates by following the instructions at https://cloud.google.com/appengine/docs/ssl. This service has a fee and we will be charged monthly.
Google Developer Console

Before Google Cloud Platform was released, Admin Console was the only tool available to developers to perform administrative and monitoring tasks on App Engine applications. Admin Console provides a lot of functionalities and it's still powerful enough to manage App Engine applications of any size. However, it's not the right tool if we extensively use the new range of services offered by the Google Cloud Platform, especially if we store data on Google Cloud Storage or our database server is Google Cloud SQL; in this case, to collect information such as billing data and usage history we have to interact with other tools.

Recently Google released Developer Console, a comprehensive tool to manage and monitor services, resources, authentication, and billing information for Google Cloud Platform, including App Engine applications. We can access the Developer Console at https://console.developers.google.com/ and log in with a valid Google user account or a Google apps account for custom domains.

To emphasize the concept that developers can combine various pieces coming from Google’s cloud infrastructure to build complex applications, Developer Console introduces the notion of cloud projects. A project is a set of functionally grouped cloud products that share the same team and billing information. At the core of a project there is always an App Engine application: every time we create a project, an App Engine application pops up in Admin Console. Simultaneously, when we register an application in Admin Console, a corresponding project is created and listed in Developer Console. Every project is identified by a descriptive name, which is a unique identifier called project ID that is also the ID of the related App Engine application and another unique identifier that is automatically generated called project number.

Beside creating and deleting projects, the developer console also let us do the following:

- **Manage project members**: When we create a project we become the owner of that project. As owners, we can add or remove members and set their permissions.
- **Manage APIs**: We can add or remove API services provided by Google Cloud Platform, set up billing, and monitor data.
- **Manage applications identity**: We can tie requests to specific projects so that we can monitor specific traffic and billing and enforce quotas if needed.
- **Manage applications security**: We can set up OAuth2 for our applications or provide API keys to authorize requests.
- **Filter and cap services**: We can allow requests coming only from authorized hosts or IP addresses and limit the amount of requests allowed for each user every second or every day for all the users.
For every service of Google Cloud Platform, Developer Console provides us with handy tools to perform maintenance operations through the web interface. For example, we can add or remove Google Cloud SQL instances, perform queries on Google Cloud Datastore, browse and manipulate the content of Google Cloud Storage, and manage virtual machines running on Google Compute Engine. We will use several parts of Developer Console later in the book.

**Development Console**

When we are on the local development server we can still access a tool to browse and manage Datastore, task queues, cron jobs, and other App Engine emulated components running locally. This tool is called Development Console and is accessible at `http://localhost:8000` when the local server is active.

**Summary**

In this chapter we have learned what Google Cloud Platform is, the tools and services it provides, and how we can use them to develop and run fast and scalable web applications written in Python.

We explored what tools we need to start developing with Python for the App Engine platform, how to run an application locally with the development server, and how fast and easy it is to upload it in a production server, ready to be served through the Internet.

The simple example we used in the chapter, although a fully functional App Engine application, is quite simple and it doesn’t make use of anything provided by the platform besides the runtime environment. In the next chapter, we will start from scratch with a new, more useful application, exploring the webapp2 framework and taking advantage of Cloud Datastore.
Where to buy this book

You can buy Python for Google App Engine from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.