Flask By Example

This book will take you on a journey from learning about web development using Flask to building fully functional web applications. In the first major project, we develop a dynamic Headlines application that displays the latest news headlines along with up-to-date currency and weather information. In project two, we build a Crime Map application that is backed by a MySQL database, allowing users to submit information on and the location of crimes in order to plot danger zones and other crime trends within an area. In the final project, we combine Flask with more modern technologies, such as Twitter’s Bootstrap and the NoSQL database MongoDB, to create a Waiter Caller application that allows restaurant patrons to easily call a waiter to their table.

This pragmatic tutorial will keep you engaged as you learn the use of Flask by working on challenging real-world applications.

Who this book is written for

This book is aimed at developers and hobbyists who have some knowledge of Python but no knowledge of the micro framework, Flask. It touches on other technologies, including MySQL, MongoDB, HTML, JavaScript, and CSS. You will find that having experience in any of these technologies will be beneficial, but only basic Python knowledge is required.

What you will learn from this book

- Build three web applications from the ground up using the powerful Python micro framework, Flask
- Dynamically display data to your viewers, based on their requests
- Store user and static data in SQL and NoSQL databases and use this data to power your web applications
- Create a good user experience by combining HTML, CSS, and JavaScript
- Harness the convenience of freely available APIs, including OpenWeatherMap, OpenExchange Rates, and bitly
- Extend your applications to build advanced functionality, such as a user account control system using Flask-Login
- Learn about web application security and defend against common attacks, such as SQL injection and XSS

Unleash the full potential of the Flask web framework by creating simple yet powerful web applications.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Hello, World!'
- A synopsis of the book’s content
- More information on Flask By Example
Gareth Dwyer first heard the phrase, "behind every no-entry sign there is a door," a couple of decades ago, and he has been looking for a counterexample ever since. He hasn't found one yet. Gareth grew up with his three siblings in Grahamstown, South Africa. There wasn't much there except some highly respected schools and a small university. Gareth had heard that school was an unpleasant and largely pointless experience, so he opted to skip it and go to the university instead. The university door had a no-entry sign on the door because it only accepted people who had gone to school. Gareth ignored the sign. He studied piano for a while but soon, he wondered if there was more to life than sitting in front of a keyboard all day. So he switched from piano to computer science, and it took him a while to realize the irony. He studied philosophy too because it was here that people never told him to stop being so argumentative.

Gareth noticed the disparagement that his philosophy and computer science departments felt towards each other, and he found it strange. He soon discovered that he wasn't the first person to see that there was room for some common ground, and he went to Europe to study computational linguistics, where he found other people who liked debating the finer points of language while talking about the three hardest problems of computer science (naming things, and off-by-one errors).

In between doodling on blank paper while listening to very knowledgeable people lecture on content that was occasionally fascinating but often soporific, Gareth has gained so-called "industry" experience with companies such as Amazon Web Services in Cape Town and MWR InfoSecurity in Johannesburg. He has several years' experience in writing, and his favorite languages are English and Python.

He discovered that writing and writing a book are not fully overlapping experiences, and the former is hardly preparation for the latter. The pages that follow would not have come into existence without the combined efforts of many people.
In theory, nothing works, but everyone knows why. In practice, everything works but no one knows why. Here, we combine theory and practice; nothing works and no one knows why!

Learning computer science must always be a combination of theory and practice; you need to know what you're doing (theory), but you also need to know how to do it (practice). My experience of learning how to create web applications was that few teachers found a sweet spot for this balance; either I read pages and pages about inheritance, virtual environments, and test-driven development, wondering how it all applied to me, or I installed a bunch of tools and frameworks and libraries and watched the magic happen with no idea how it worked.

What follows is, I hope, a good balance. From the first chapter, you'll have a Flask web application running that the whole world can visit, which is quite practical even if it doesn't do anything but greet visitors with "Hello, World!". In the chapters that follow, we'll walk through building three interesting and useful projects together. In general, we'll build things ourselves wherever possible. While it's not good to reinvent the wheel, it is good to be exposed to a problem before you're exposed to the solution. Learning a CSS framework before you write a single line of CSS leaves you in a confused state, in which you would wonder, "But why do I actually need this?", and the same goes for many other frameworks and tools. So, we'll start from scratch, take a look at why it's difficult, and then introduce tools to make our lives easier. I think this is the ideal balance between theory and practice.

When I told people I was writing a book on Flask, the common response was "Why? There are already so many books and tutorials on Flask." This is a valid question, and the answer to it provides a nice outline for what to expect from this book. *Flask By Example* is different from other Flask educational material and here's why.
Preface

We won't leave you stranded

Many Flask tutorials show you how to develop a Flask application and run it locally on your own machine, and then they end. This is great as a first step, but if you're interested in building web applications, you probably want them to be accessible on the Web so that your friends, family, coworkers, and customers can admire your handiwork without popping by your house. From our first project onward, our applications will run on a Virtual Private Server (VPS) and be accessible to the world.

We won't build a blogging application

If you've read any web application development tutorials, you must have noticed that nearly every one of them is about how to build a blog using x and y. I'm pretty tired of the blog example (actually, I never want to see anyone show me how to build a blog again). Instead, you'll create some interesting, original, and possibly even useful projects while learning how to develop web applications with Flask.

We will focus on security

Cybercrime has become something of a buzzword of late. Arguably, the reason that we read about major web applications being hacked on an almost daily basis is because so many developers do not know about SQL Injection, CSRF, XSS, how to store passwords, and so many other things that should really be considered basic knowledge. As we develop the three projects in this book, we'll take the time to explain some core security concepts in detail and show you how to harden our applications against potentially malicious attackers.

We will give in-depth explanations

We won't just give you some code and tell you to run it. Wherever possible, we will explain what we're doing, why we're doing it, and how we're doing it. This means that you'll be able to take ideas from all of the projects, combine them with your own ideas, and get started with building original content right after working through this book.

Therefore, I hope that this book will be of use to you, no matter whether you are beginning to cut your teeth in the world of computer science and programming or have a computer science degree from a famous university and have compiler theory pouring out of your ears but now want to build something practical and fun. May you have as much fun working through the projects as I did while putting them together!
What this book covers

Chapter 1, Hello, World!, teaches you to set up our development environment and a web server and write our first Flask application.

Chapter 2, Getting Started with Our Headlines Project, shows you how to run Python code when the user visits a URL and how to return basic data to the user. We will also look at fetching the latest headlines automatically using RSS feeds.

Chapter 3, Using Templates in Our Headlines Project, introduces Jinja templates and integrates them into our Headlines project. We will show how to serve dynamic HTML content by passing data from our Python code to template files.

Chapter 4, User Input for Our Headlines Project, shows how to get input from our users over the Internet and use this input to customize what we will show our users. We will look at how to access current weather information through JSON APIs and include this information in our Headlines project.

Chapter 5, Improving the User Experience of Our Headlines Project, instructs you to add cookies to our Headlines project so that our application can remember our users' choices. We will also style our application by adding some basic CSS.

Chapter 6, Building an Interactive Crime Map, introduces our new project, which is a crime map. We will introduce relational databases, install MySQL on our server, and look at how to interact with our database from our Flask application.

Chapter 7, Adding Google Maps to our Crime Map Project, instructs you on adding a Google Maps widget and shows how to add and remove markers from the map based on our database. We will add an HTML form with various inputs for users to submit new crimes and also display the existing crimes.

Chapter 8, Validating User Input in Our Crime Map Project, polishes off our second project by making sure that users can't break it accidentally or through maliciously crafted input.

Chapter 9, Building a Waiter Caller App, introduces our final project, which is an application to call a waiter to the table at a restaurant. We will introduce Bootstrap and set up a basic User Account Control system that uses Bootstrap as the frontend.

Chapter 10, Template Inheritance and WTForms in Waiter Caller Project, introduces Jinja's template inheritance features so that we can add similar pages without duplicating code. We will use the WTForms library to make our web forms easier to build and validate.
Preface

Chapter 11, Using MongoDB with Our Waiter Caller Project, discusses how to install and configure MongoDB on our server and links it to our Waiter Caller project. We will finish off our final project by adding indices to our database and a favicon to our application.

Appendix, A Sneak Peek into the Future, outlines some important topics and technologies that we weren't able to cover in detail and gives pointers on where more can be learned about these.
Hello, World!

And hello, reader! Let's get started with building some Flask applications. Flask is minimalist enough to allow you choice and flexibility; unlike in larger frameworks, you choose what you want to do and then manipulate Flask to do your bidding, and it is complete enough to work right out of the box.

We'll walk together through the development of three web applications; the first one is straightforward and will allow you to cut your teeth on Flask and get used to the new technologies and terminology while building a nontrivial web application; the second will get you started with building a web application that makes use of a traditional SQL database; and the final, which has the most features, will make use of a NoSQL database and a frontend framework to create a useful and good-looking web application.

In this chapter, we'll take a brief look at what Flask is and, perhaps more importantly, what it isn't. We'll move on to setting up our basic development environment as well as a web server, and we'll install a Python package manager as well as Flask itself. By the end of the chapter, we'll have the outlines of our first app, and, as dictated by age-old tradition, we'll use our new skills to display the text "Hello, World!".

In brief, we will cover the following topics:

- Introducing Flask
- Creating our development environment
- Writing "Hello, World!"
- Deploying our application to production
Introducing Flask

Flask is a micro framework for Python web development. A framework, in the simplest terms, is a library or collection of libraries that aims to solve a part of a generic problem instead of a complete specific one. When building web applications, there are some problems that will always need to be solved, such as routing from URLs to resources, inserting dynamic data into HTML, and interacting with an end user.

Flask is a micro framework because it implements only core functionality (including routing) but leaves more advanced functionality (including authentication and database ORMs) to extensions. The result of this is less initial setup for the first-time user and more choice and flexibility for the experienced user. This is in contrast with "fuller" frameworks, such as Django, which dictate their own ORM and authentication technologies.

As we'll discuss, our Hello World application in Flask can be written in only seven lines of code, with the entire application consisting of a single file. Does that sound good? Let's get going!

Creating our development environment

A development environment consists of all the software that a developer uses while building software. For starters, we'll install a Python package manager (pip) and the Flask package. In this book, we'll show detailed steps for developing using Python 2.7 on a clean installation of Ubuntu 14.04, but everything should be easy to translate to Windows or OS X.

Installing pip

For our Hello World application, we only need the Python Flask package, but we'll install several Python packages throughout the development process of our three applications. To manage these packages, we'll use the Python package manager pip. If you've developed in Python without a package manager until now, you'll love how easy it is to download, install, remove, and update packages using pip. If you already use it, then skip to the next step where we'll use it to install Flask.

The pip manager is included in Python's 3.4+ and 2.7.9+ versions. For older versions of Python, pip needs to be installed. To install pip on Ubuntu, open a terminal and run the following command:

```bash
sudo apt-get update
sudo apt-get install python-pip
```
To install pip on Windows or OS X, you can download and run the get-pip.py file from the pip homepage at https://pip.pypa.io/en/latest/installing/#install-or-upgrade-pip.

That's it! You can now easily install any Python package you need through pip.

**Installing Flask**

Installing Flask through pip could not be more straightforward. Simply run the following:

```
pip install --user flask
```

You might see some warnings in your terminal, but at the end, you should also see **Successfully installed Flask**. Now, you can import Flask into a Python program just as with any other library.

If you're used to using VirtualEnv for Python development, you can install Flask inside a VirtualEnv environment. We will discuss this further in *Appendix, A Sneak Peek into the Future*.

**Writing "Hello, World!"**

Now, we'll create a basic web page and serve it using Flask's built-in server to localhost. This means that we'll run a web server on our local machine that we can easily make requests to from our local machine. This is very useful for development but not suited for production applications. Later on, we'll take a look at how to serve Flask web applications using the popular Apache web server.

**Writing the code**

Our application will be a single Python file. Create a directory in your home directory called `firstapp` and a file inside this called `hello.py`. In the `hello.py` file, we'll write code to serve a web page comprising the static string "Hello, World!". The code looks as follows:

```python
from flask import Flask

app = Flask(__name__)
```
Hello, World!

```python
@app.route("/")
def index():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

**Downloading the example code**

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

- Log in or register to our website using your e-mail address and password.
- Hover the mouse pointer on the SUPPORT tab at the top.
- Click on Code Downloads & Errata.
- Enter the name of the book in the Search box.
- Select the book for which you're looking to download the code files.
- Choose from the drop-down menu where you purchased this book from.
- Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipex / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

Let's break down what this does. The first line should be familiar; it simply imports Flask from the package flask. The second line creates an instance of the Flask object using our module's name as a parameter. Flask uses this to resolve resources, and in complex cases, one can use something other than `__name__` here. For our purposes, we'll always use `__name__`, which links our module to the Flask object.

Line 3 is a Python decorator. Flask uses decorators for URL routing, so this line of code means that the function directly below it should be called whenever a user visits the main root page of our web application (which is defined by the single forward slash). If you are not familiar with decorators, these are beautiful Python shortcuts that seem a bit like black magic at first. In essence, they call a function that takes the function defined under the decorator (in our case, `index()`) and returns a modified function.
The next two lines should also seem familiar. They define a very simple function that returns our message. As this function is called by Flask when a user visits our application, the return value of this will be what is sent in response to a user who requests our landing page.

Line 6 is a Python idiom with which you are probably familiar. This is a simple conditional statement that evaluates to True if our application is run directly. It is used to prevent Python scripts from being unintentionally run when they are imported into other Python files.

The final line kicks off Flask's development server on our local machine. We set it to run on port 5000 (we'll use port 80 for production) and set debug to True, which will help us see detailed errors directly in our web browser.

Running the code
To run our development web server, simply fire up a terminal and run the hello.py file. If you used the same structure outlined in the previous section, the commands will be as follows:

cd firstapp/hello
python hello.py

You should get an output similar to that in the following screenshot:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
```

Also, you should see the process continue to run. This is our web server listening for requests. So, let's make a request!

Fire up a web browser—I use Firefox, which comes packaged with Ubuntu—and navigate to localhost:5000.

The localhost part of the URL is a shortcut to the loopback address, usually 127.0.0.1, which asks your computer to make the web request to itself. The number after the colon (5000) is the port it should make the request to. By default, all HTTP (web) traffic is carried over port 80. For now, we'll use 5000 as it is unlikely to conflict with any existing services, but we'll change over to port 80 in production, which is conventional, so that you won't have to worry about the colon.
Hello, World!

You should see the "Hello, World!" string displayed in your browser as in the following screenshot. Congratulations, you've built your first web application using Flask!

![Hello, World!](image)

Deploying our application to production

It's great to have an application that runs, but inherent to the idea of a web application is the idea that we want others to be able to use it. As our application is Python-based, we are a bit limited in how we can run our application on a web server (many traditional web hosts are only configured to run PHP and/or .NET applications). Let's consider how to serve Flask applications using a **Virtual Private Server (VPS)** running Ubuntu Server, Apache, and WSGI.

From this point on, we'll maintain two environments. The first is our development environment, which we just set up and where we'll write code and view its results using the Flask server running on localhost (as we just did). The second will be a production environment. This will be a server to which we can deploy our web applications and make them accessible to the world. When we install new Python libraries or other software on our development environment, we'll normally want to mirror our actions in the production environment.

**Setting up a Virtual Private Server**

Although you could, in theory, host your web application on your local machine and allow others to use it, this has some severe limitations. First of all, every time you turned off your computer, your app would not be available. Also, your computer probably connects to the Internet via an Internet Service Provider (ISP) and possibly a wireless router. This means that your IP address is dynamic and changes regularly, which makes it difficult for your applications' users to keep up! Finally, chances are that you have an asymmetrical connection to the Internet, which means that your upload speed is slower than your download speed.
Hosting your application on a server solves all of these problems. Before "the cloud" became popular, the traditional way to host a web application was to buy a physical server and find a data center to host it. These days, things are far simpler. In a few minutes, you can fire up a virtual server, which to you seems just like a physical server—you can log in to it, configure it, and enjoy full control over it—but it is actually just a virtual "piece" of a machine owned and controlled by a cloud provider.

At the time of writing, major players in the cloud provider field include Amazon Web Services, Microsoft Azure, Google Cloud Compute, and Digital Ocean. All of these companies allow you to hire a virtual server or servers upon paying by the hour. If you are learning Flask as a hobby and are unwilling to pay anyone to host your web applications, you'll probably find a free trial at one of the providers quite easily. The smallest offering by any provider is fine to host all the applications that we'll run.

Select one of the preceding providers or another of your choosing. If you've never done anything similar before, Digital Ocean is often cited to have the simplest process of signing up and creating a new machine. Once you select a provider, you should be able to follow their respective instructions to fire up a VPS that runs Ubuntu Server 14.04 and SSH into it. You'll have full control over the machine with one slight difference: you won't have a display or a mouse.

You'll enter commands on your local terminal, which will in fact be run on the remote machine. Detailed instructions on how to connect to your VPS will be given by the provider, but if you use Ubuntu, it should be as simple as running the following:

```bash
ssh user@123.456.789.000
```

Alternatively, if you set it up with a public-private key authentication, where `yourkey.pem` is the full path to your private key file, here's the command to run:

```bash
ssh user@123.456.78.000 -i yourkey.pem
```

Here, `user` is the default user on the VPS, and `yourkey` is the name of your private key file.

**SSH from other operating systems:**

SSH from OS X should be the same as Ubuntu, but if you're using Windows, you'll have to download PuTTY. Refer to [http://www.putty.org/](http://www.putty.org/) to download and for full usage instructions. Note that if you use key files for authentication, you'll have to convert them to a format compatible with PuTTY. A conversion tool can also be found on the PuTTY website.
Once we connect to the VPS, installing Flask is the same process as it was previously:

```sh
sudo apt-get update
sudo apt-get install python-pip
pip install --user Flask
```

To install our web server, Apache, and WSGI, we will run the following:

```sh
sudo apt-get install apache2
sudo apt-get install libapache2-mod-wsgi
```

Apache is our web server. It will listen for web requests (which are generated by our users visiting our web application using their browsers) and hand these requests over to our Flask application. As our application is in Python, we also need WSGI (Web Server Gateway Interface).

This is a common interface between web servers and Python applications, which allows Apache to talk to Flask and vice versa. An overview of the architecture can be seen in the following diagram:

![Architecture Diagram](image)

### Configuring our server

Now that we've installed Apache, we can see our first results. You're probably used to visiting websites using a URL, such as `http://example.com`. We'll access our web applications using the IP address of our VPS directly. Your VPS should have a static public address. Static means that it doesn't change periodically, and public means that it is globally unique. When you connected to the VPS via SSH, you probably used the public IP to do this. If you can't find it, run the following on your VPS and you should see an `inet addr` section in the output, which contains your public IP:

```sh
ifconfig
```
The IP address should look similar to 123.456.78.9. Enter your IP address into your browser's address bar, and you should see a page saying "Apache2 Ubuntu Default Page: It Works!" or something similar, as in the following screenshot:

![Apache2 Ubuntu Default Page](image_url)

This means that we can now serve web content to anyone with an Internet connection! However, we still have to:

- Copy our code to the VPS
- Link up Apache and Flask
- Configure Apache to serve our Flask application

For the first step, we'll set up a Git repository on our local machine and clone the repository to the VPS. For the second step, we'll use the WSGI module that we installed with Apache. Finally, we'll take a look at how to write a virtual host to have Apache serve our Flask application by default.

### Installing and using Git

Git is a version control system. A version control system, among other things, saves multiple versions of our code base automatically. This is great to undo accidental changes or even deletions; we can simply revert to a previous version of our code. It also includes lots of functionality for distributed development—that is, many developers working on a single project. We'll use it mainly for its backup and deployment features, however.

To install Git on your local machine and VPS, run the following commands on each:

```bash
sudo apt-get update
sudo apt-get install git
```
Make sure you're comfortable with the difference between running commands on your own machine using the terminal and on your server through the SSH connection. In many cases, we'll need to run the same commands twice—once for each environment.

Now that you have the software, you need a place to host your Git repositories or "repos". Two popular and free Git hosting services are GitHub (http://github.com) and Bitbucket (http://bitbucket.org). Head over to one of them, create an account, and create a new repository by following the instructions that they provide. When given the option to give your repository a name, call it firstapp to match the name of the directory that we will use for our code base. Once you create a new repository, you should be given a unique URL to your repository. Take note of this as we'll use it to push our Hello, World! application using git and then deploy it to our VPS.

On your local machine, open a terminal and change the directory to the Flask application. Initialize a new repository and link it to your remote Git repository via the following commands:

```
cd firstapp
git init
git remote add origin <your-git-url>
```

Tell git who you are, to allow it to automatically add metadata to your code changes, as follows:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Git allows you full control over which files are part of your repository and which aren't. Even though we initialized the Git repo in our firstapp directory, our repo currently contains no files. Add our application to the repo, commit, and then push it, as follows:

```
git add hello.py
```
```
git commit -m "Initial commit"
```
```
git push -u origin master
```
These are the main Git commands that we'll use throughout this book, so let's take a brief look at what each does. The add command adds new or modified files to our repository. This tells Git which files are actually part of our project. Think of the commit command as taking a snapshot of our project in its current state. This snapshot is saved on our local machine. It is good to make a new commit with any major change to the code base as we can easily revert to previous commits if a later commit breaks our application. Finally, the push command pushes our local changes to the remote Git server. This is good for backup, and it will also allow us to fetch the changes on our VPS, thus keeping the code base on our local machine and that on our VPS in sync.

Now, SSH into your VPS again and get a copy of our code, as follows:

```bash
cd /var/www
git clone <your-git-url>
```

Where the `<your-git-url>` part of the above command is actually a placeholder for the URL to your Git repository.

If you get a permission denied error on trying to clone the Git repository, you might need to take ownership of the `/var/www` directory for the Linux user that you're using. If you logged into the server with `tom@123.456.789.123`, you can run the following command, which will give your user ownership of `/var/www` and allow you to clone the Git repository into it. Again `tom` is the placeholder used in the following case:

```bash
sudo chown -R tom /var/www
```

If you used `firstapp` as a name for your remote repository, this should create a new directory called `firstapp`. Let's verify that our code is there using the following:

```bash
cd firstapp
ls
```

You should see your `hello.py` file. Now, we need to configure Apache to use WSGI.

## Serving our Flask app with WSGI

First, we'll create a very straightforward `.wsgi` file in our application directory. Then, we'll create an Apache configuration file in the directory where Apache looks for available sites.
The only slightly tricky part about these two steps is that we'll create the files directly on our VPS, and as our VPS does not have a display, this means that we have to use command-line interface text editors. Of course, we could create the files locally and then transfer them to our VPS as we did for our code base, but for small changes to configuration files, this is often more effort than it's worth. Using a text editor without a mouse takes a bit of getting used to, but it's a good skill to learn. The default text editor on Ubuntu is Nano, and the other popular choices are vi or Vim. Some people use Emacs. If you already have a favorite, go with it. If not, we'll use Nano for the examples in this book (it is already installed and arguably the simplest to use). However, if you want to go the extra mile, I recommend learning to use Vim.

Assuming you're still connected to your VPS and have navigated to the /var/www/firstapp directory as in the most recent steps, run the following command:

```
nano hello.wsgi
```

This creates the hello.wsgi file, which you can now edit through Nano. Type the following:

```python
import sys
sys.path.insert(0, "/var/www/firstapp")
from hello import app as application
```

This is simply Python syntax, which patches our application into the PATH system so that Apache can find it through WSGI. We will then import app (we named this in our hello.py app with the `app = Flask(__name__)` line) into the namespace.

Hit Ctrl + X to exit Nano and enter Y when prompted to save the changes.

Now, we'll create an Apache configuration file that points to the .wsgi file we just created, as follows:

```
cd /etc/apache2/sites-available
nano hello.conf
```

If you run into permission issues while editing or saving files, you may need to take ownership of the apache2 directory too. Run the following command, substituting the username for your Linux user:

```
sudo chown -R tom /etc/apache2
```

In this file, we'll create a configuration for an Apache virtual host. This will allow us to serve multiple sites from a single server, which will be useful later when we want to serve other applications using our single VPS. In Nano, enter the following configuration:
<VirtualHost *>
  ServerName example.com

  WSGIScriptAlias / /var/www/firstapp/hello.wsgi
  WSGIDaemonProcess hello
  <Directory /var/www/firstapp>
    WSGIProcessGroup hello
    WSGIApplicationGroup %{GLOBAL}
    Order deny,allow
    Allow from all
  </Directory>
</VirtualHost>

This might look quite complicated, but it's actually very straightforward. We will create a virtualhost and specify our domain name, where our .wsgi script is, the name of our application, and who is allowed to access it. We'll discuss domain names in the final chapter, but for now, you can just leave it as example.com because we'll access our application by its IP address.

If you get stuck on this step, the Flask website has a great resource on configuring and troubleshooting Apache configuration. You can find it at http://flask.pocoo.org/docs/0.10/deploying/mod_wsgi/.

Hit Ctrl + X and enter Y when prompted again to save and exit the file.

Now, we need to enable the configuration and set it as our default site.

**Configuring Apache to serve our Flask application**

Apache sites work as follows: there is a sites-available directory (where we created the new virtual host configuration file) and a sites-enabled directory, which contains shortcuts to all the configuration files that we want to be active. By default, you'll see a file in the sites-available directory named 000-default.conf. This is the reason that we saw a default It works Apache page when we first installed Apache. We don't want this anymore; instead, we want to use our application as the default site. Therefore, we'll disable the default Apache site, enable our own, and then restart Apache for the changes to take effect. Run these commands to do this:

```bash
sudo a2dissite 000-default.conf
sudo a2ensite hello.conf
sudo service apache2 reload
```
The required Apache configuration and commands can vary quite a bit based on the platform you're using. If you use Ubuntu Server as recommended, the preceding should all work smoothly. If not, you may need to read up a bit on how to configure Apache for your specific platform.

You should note reloading web server apache2 in the output. If errors are displayed, then you probably misconfigured something in the preceding command. If this is the case, read the error message carefully and go back over the previous steps to take a look at why things didn't work as expected.

To test that everything is working, open a web browser on your local machine and type your IP address into the address bar again. You should see **Hello, World!** displayed in your browser instead of the default Apache page that we saw before.

If you get **Error 500** instead, it means that our application fell over for some reason. Fear not; it's better that you get used to dealing with this error now, when the fix will probably be simple, than later on, when we've added more components that could break or be misconfigured. To find out what went wrong, run the following command on your VPS:

```
sudo tail -f /var/log/apache2/error.log
```

The `tail` command simply outputs the last several lines of the file passed as an argument. The `-f` is for follow, which means that the output will be updated if the file changes. If you can't immediately work out which lines are indicative of the error we're looking for, visit the site in your web browser on your local machine again, and you'll see the output from the `tail` command be updated accordingly. The following screenshot shows the output from the `tail` command when there are no errors; however, if anything goes wrong, you'll see the error output printed among all the info messages.
Some possible tripping points are incorrectly configured WSGI and Apache files (make sure that your WSGIDaemonProcess and daemon name match, for example) or incorrectly configured Python (you may forget to install Flask on your VPS). If you can't figure out what the error message means, an Internet search for the message (removing the error-specific parts of your app, such as names and paths) will usually point you in the right direction. Failing this, there are strong and very friendly Flask and WSGI communities on Stack Overflow and Google Groups, and there's normally someone willing to help beginners. Remember that if you're having a problem and can't find an existing solution online, don't feel bad for asking; you'll help countless people facing issues similar to yours.
Summary

We got through quite a lot of material in this first chapter! We did some initial setup and house-keeping and then wrote our first web application using Flask. We saw this run locally and then discussed how to use Git to copy our code to a server. We configured our server to serve our application to the public; however, our application is merely a static page that prints the "Hello, World!" string to whoever visits our page. This is not useful to many people and could be achieved more simply using a static HTML page. However, with the extra effort we put in, we now have all the power of Python behind our application; we're just not using it yet!

In the next chapter, we'll discover how to take advantage of Python to make our web applications more useful!
Where to buy this book
You can buy Flask By Example 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.