Chapter No. 1
"Flask Configurations"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.1 "Flask Configurations"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Shalabh Aggarwal has several years of experience in developing business systems and web applications for small-to-medium scale industries. He started his career working on Python, and although he works on multiple technologies, he remains a Python developer at heart. He is passionate about open source technologies and writes highly readable and quality code. He is a major contributor to some very popular open source applications. He has worked with Openlabs Technologies and Consulting (P) Limited as the CTO for a large part of his career. He is also active in voluntary training for engineering students on nonconventional and open source topics.

When not working with full-time assignments, he consults for start-ups on leveraging different technologies. When not writing code, he writes non-technical literature and makes music with his guitar.

I would like to dedicate this book to my late father who will always be there in my thoughts for the love and encouragement he gave me to explore new things in life. I would like to thank my family, my mother and my sister, for putting up with me during my long writing and research sessions. I would also like to thank my friends and colleagues who encouraged me and kept the momentum going. I would like to convey deep gratitude to my mentor, Sharoon Thomas, who introduced me to these technologies and helped me learn a lot. Without the support of all of them, I would have never been able to learn these technologies and complete this book.

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
Flask Framework Cookbook

Flask is a lightweight web application microframework written in Python. It makes use of the flexibility of Python to provide a relatively simple template for web application development. Flask makes it possible to write simple one-page applications, but it also has the power to scale them and build larger applications without any issues.

Flask has excellent documentation and an active community. It has a number of extensions, each of which have documentation that can be rated from good to excellent. There are a few books also available on Flask; they are great and provide a lot of insight into the framework and its applications. This book tries to take a different approach to explain the Flask framework and multiple aspects of its practical uses and applications as a whole.

This book takes you through a number of recipes that will help you understand the power of Flask and its extensions. You will start by seeing the different configurations that a Flask application can make use of. From here, you will learn how to work with templates, before learning about the ORM and view layers, which act as the foundation of web applications. Then, you will learn how to write RESTful APIs with Flask, after learning various authentication techniques. As you move ahead, you will learn how to write an admin interface followed by the debugging and logging of errors in Flask. You will also learn how to make your applications multilingual and gain an insight into the various testing techniques. Finally, you will learn about the different deployment and post-deployment techniques on platforms such as Apache, Tornado, Heroku, and AWS Elastic Beanstalk.

By the end of this book, you will have all the necessary information required to make the best use of this incredible microframework to write small and big applications and scale them with industry-standard practices.

A good amount of research coupled with years of experience has been used to develop this book, and I really wish that this book will benefit fellow developers.

What This Book Covers

Chapter 1, Flask Configurations, helps in understanding the different ways in which Flask can be configured to suit various needs as per the demands of the project. It starts by telling us how to set up our development environment and moves on to the various configuration techniques.

Chapter 2, Templating with Jinja2, covers the basics of Jinja2 templating from the perspective of Flask and explains how to make applications with modular and extensible templates.

For More Information:
www.packtpub.com/web-development/flask-framework-cookbook
Chapter 3, *Data Modeling in Flask*, deals with one of the most important parts of any application, that is, its interaction with the database systems. We will see how Flask can connect to database systems, define models, and query the databases for the retrieval and feeding of data.

Chapter 4, *Working with Views*, talks about how to interact with web requests and the proper responses to be catered for these requests. It covers various methods of handling the requests properly and designing them in the best way.

Chapter 5, *Webforms with WTForms*, covers form handling, which is an important part of any web application. As much as the forms are important, their validation holds equal importance, if not more. Presenting this information to the users in an interactive fashion adds a lot of value to the application.

Chapter 6, *Authenticating in Flask*, deals with authentication, which sometimes acts as a thin red line between the application being secure and insecure. This chapter deals with social logins in detail.

Chapter 7, *RESTful API Building*, helps in understanding REST as a protocol and then talks about writing RESTful APIs for Flask applications.

Chapter 8, *Admin Interface for Flask Apps*, focuses on writing admin views for Flask applications. First, we will write completely custom-made views and then write them with the help of an extension.

Chapter 9, *Internationalization and Localization*, expands the scope of Flask applications and covers the basics of how to enable support for multiple languages.

Chapter 10, *Debugging, Error Handling, and Testing*, moves on from being completely development-oriented to testing our application. With better error handling and tests, the robustness of the application increases manifold and debugging aids in making the lives of developers easy.

Chapter 11, *Deployment and Post Deployment*, covers the various ways and tools using which the application can be deployed. Then, you will learn about application monitoring, which helps in keeping track of the performance of the application.

Chapter 12, *Other Tips and Tricks*, is a collection of some handy tricks that range from full-text search to caching. Then finally, we will go asynchronous with certain tasks in Flask applications.

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
Flask Configurations

This introductory chapter will help you to understand the different ways Flask can be configured to suit various needs as per the demands of the project.

In this chapter, we will cover the following recipes:

- Environment setup with virtualenv
- Handling basic configurations
- Class-based settings
- Organization of static files
- Being deployment specific with instance folders
- Composition of views and models
- Creating a modular web app with blueprints
- Making a Flask app installable using setuptools

Introduction

"Flask is a microframework for Python based on Werkzeug, Jinja2 and good intentions."

Flask official documentation

Why micro? Does it mean that Flask is lacking in functionality or that your complete web application has to mandatorily go inside one file? Not really! It simply refers to the fact that Flask aims at keeping the core of the framework small but highly extensible. This makes writing applications or extensions very easy and flexible and gives developers the power to choose the configurations they want for their application, without imposing any restrictions on the choice of database, templating engine, and so on. In this chapter, you will learn some ways to set up and configure Flask.

For More Information:
www.packtpub.com/web-development/flask-framework-cookbook
Getting started with Flask hardly takes 2 minutes. Setting up a simple Hello World application is as easy as baking a pie:

```python
from flask import Flask
app = Flask(__name__)

@app.route('/
    def hello_world():
        return 'Hello to the World of Flask!'"n
if __name__ == '__main__':
    app.run()
```

Now, Flask needs to be installed; this can be done simply via pip:

```
$ pip install Flask
```

The preceding snippet is a complete Flask-based web application. Here, an instance of the imported Flask class is a Web Server Gateway Interface (WSGI) application. So, app in this code becomes our WSGI application, and as this is a standalone module, we set the `__name__` string as `'__main__'`. If we save this in a file with the name `app.py`, then the application can simply be run using the following command:

```
$ python app.py
* Running on http://127.0.0.1:5000/
```

Now, if we just head over to our browser and type http://127.0.0.1:5000/, we can see our application running.

Never save your application file as `flask.py`; if you do so, it will conflict with Flask itself while importing.

**Environment setup with virtualenv**

Flask can be installed using `pip` or `easy_install` globally, but we should always prefer to set up our application environment using `virtualenv`. This prevents the global Python installation from getting affected by our custom installation by creating a separate environment for our application. This separate environment is helpful because you can have multiple versions of the same library being used for multiple applications, or some packages might have different versions of the same libraries as dependencies. `virtualenv` manages this in separate environments and does not let a wrong version of any library affect any application.

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
**How to do it...**

We will first install `virtualenv` using `pip` and then create a new environment with the name `my_flask_env` inside the folder in which we ran the first command. This will create a new folder with the same name:

```
$ pip install virtualenv
$ virtualenv my_flask_env
```

Now, from inside the `my_flask_env` folder, we will run the following commands:

```
$ cd my_flask_env
$ source bin/activate
$ pip install flask
```

This will activate our environment and install Flask inside it. Now, we can do anything with our application within this environment, without affecting any other Python environment.

**How it works...**

Until now, we have used `pip install flask` multiple times. As the name suggests, the command refers to the installation of Flask just like any Python package. If we look a bit deeper into the process of installing Flask via `pip`, we will see that a number of packages are installed. The following is a summary of the package installation process of Flask:

```
$ pip install -U flask
Downloading/unpacking flask
.........
.........
Many more lines.........
.........
Successfully installed flask Werkzeug Jinja2 itsdangerous markupsafe
Cleaning up...
```

In the preceding command, `-U` refers to the installation with upgrades. This will overwrite the existing installation (if any) with the latest released versions.

If we notice carefully, there are five packages installed in total, namely `flask`, `Werkzeug`, `Jinja2`, `itsdangerous`, and `markupsafe`. These are the packages on which Flask depends, and it will not work if any of them are missing.

---

**For More Information:**

There's more...

To make our lives easier, we can use virtualenvwrapper, which, as the name suggests, is a wrapper written over virtualenv and makes the handling of multiple virtualenv easier.

Remember that the installation of virtualenvwrapper should be done at a global level. So, deactivate any virtualenv that might still be active. To deactivate it, just use the following command:

```
$ deactivate
```

Also, it is possible that you might not be able to install the package at a global level because of permission issues. Switch to superuser or use `sudo` in this case.

You can install virtualenvwrapper using the following commands:

```
$ pip install virtualenvwrapper
$ export WORKON_HOME=~/.workspace
$ source /usr/local/bin/virtualenvwrapper.sh
```

In the preceding code, we installed virtualenvwrapper, created a new environment variable with the name WORKON_HOME, and provided it with a path, which will act as the home for all our virtual environments created using virtualenvwrapper. To install Flask, use the following commands:

```
$ mkvirtualenv flask
$ pip install flask
```

To deactivate a virtualenv, we can just run the following command:

```
$ deactivate
```

To activate an existing virtualenv using virtualenvwrapper, we can run the following command:

```
$ workon flask
```

See also

References and installation links are as follows:

- https://pypi.python.org/pypi/virtualenv
- https://pypi.python.org/pypi/virtualenvwrapper
- https://pypi.python.org/pypi/Flask
- https://pypi.python.org/pypi/Werkzeug

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
Handling basic configurations

The first thing that comes to mind is configuring a Flask application as per the need. In this recipe, we will try to understand the different ways in which Flask configurations can be done.

Getting ready

In Flask, a configuration is done on an attribute named `config` of the `Flask` object. The `config` attribute is a subclass of the dictionary data type, and we can modify it just like any dictionary.

How to do it...

For instance, to run our application in the debug mode, we can write the following:

```python
app = Flask(__name__)
app.config['DEBUG'] = True
```

The debug Boolean can also be set at the `Flask` object level rather than at the `config` level:

```python
app.debug = True
```

Alternatively, we can use this line of code:

```python
app.run(debug=True)
```

Enabling the debug mode will make the server reload itself in the case of any code changes, and it also provides the very helpful Werkzeug debugger when something goes wrong.

There are a bunch of configuration values provided by Flask. We will come across them in the relevant recipes.

As the application grows larger, there originates a need to manage the application's configuration in a separate file as shown here. Being specific to machine-based setups in most cases will most probably not be a part of the version-control system. For this, Flask provides us with multiple ways to fetch configurations. The most frequently used ones are discussed here:

- From a Python configuration file (*.cfg), the configuration can be fetched using:

  ```python
  app.config.from_pyfile('myconfig.cfg')
  ```

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
**Flask Configurations**

- From an object, the configuration can be fetched using:
  ```python
  app.config.from_object('myapplication.default_settings')
  ```

  Alternatively, we can also use:
  ```python
  app.config.from_object(__name__) #To load from same file
  ```

- From the environment variable, the configuration can be fetched using:
  ```python
  app.config.from_envvar('PATH_TO_CONFIG_FILE')
  ```

**How it works...**

Flask is intelligent enough to pick up only those configuration variables that are written in uppercase. This allows us to define any local variables in our configuration files/objects and leave the rest to Flask.

The best practice to use configurations is to have a bunch of default settings in app.py or via any object in our application itself and then override the same by loading it from the configuration file. So, the code will look like this:

```python
app = Flask(__name__)
DEBUG = True
TESTING = True
app.config.from_object(__name__)
app.config.from_pyfile('/path/to/config/file')
```

**Class-based settings**

An interesting way of laying out configurations for different deployment modes, such as production, testing, staging, and so on, can be cleanly done using the inheritance pattern of classes. As the project gets bigger, you can have different deployment modes such as development, staging, production, and so on, where each mode can have several different configuration settings, and some settings will remain the same.

**How to do it...**

We can have a default setting base class, and other classes can inherit this base class and override or add deployment-specific configuration variables.

Chapter 1

The following is an example of our default setting base class:

```python
class BaseConfig(object):
    'Base config class'
    SECRET_KEY = 'A random secret key'
    DEBUG = True
    TESTING = False
    NEW_CONFIG_VARIABLE = 'my value'

class ProductionConfig(BaseConfig):
    'Production specific config'
    DEBUG = False
    SECRET_KEY = open('/path/to/secret/file').read()

class StagingConfig(BaseConfig):
    'Staging specific config'
    DEBUG = True

class DevelopmentConfig(BaseConfig):
    'Development environment specific config'
    DEBUG = True
    TESTING = True
    SECRET_KEY = 'Another random secret key'
```

The secret key is stored in a separate file because, for security concerns, it should not be a part of your version-control system. This should be kept in the local filesystem on the machine itself, whether it is your personal machine or a server.

**How it works...**

Now, we can use any of the preceding classes while loading the application's configuration via `from_object()`. Let's say that we save the preceding class-based configuration in a file named `configuration.py`:

```python
app.config.from_object('configuration.DevelopmentConfig')
```

So, overall, this makes the management of configurations for different deployment environments flexible and easier.

**Downloading the example code**

You can download the example code files for all Packt books you have purchased 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.
Organization of static files

Organizing static files such as JavaScript, stylesheets, images, and so on efficiently is always a matter of concern for all web frameworks.

How to do it...

Flask recommends a specific way to organize static files in our application:

```
my_app/
    - app.py
    - config.py
    - __init__.py
    - static/
        - css/
        - js/
        - images/
            - logo.png
```

While rendering them in templates (say, the logo.png file), we can refer to the static files using the following line of code:

```
<img src='/static/images/logo.png'>
```

How it works...

If there exists a folder named static at the application's root level, that is, at the same level as app.py, then Flask will automatically read the contents of the folder without any extra configuration.

There's more...

Alternatively, we can provide a parameter named static_folder to the application object while defining the application in app.py:

```
app = Flask(__name__, static_folder='/path/to/static/folder')
```

In the img src path in the How to do it... section, static refers to the value of static_url_path on the application object. This can be modified as follows:

```
app = Flask(__name__,
            static_url_path='/differentstatic',
            static_folder='/path/to/static/folder'
)
```

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
Now, to render the static file, we will use the following:

```html
<img src='/differentstatic/logo.png'>
```

It is always a good practice to use `url_for` to create the URLs for static files rather than explicitly define them:

```html
<img src='{{ url_for('static', filename="logo.png") }}'>
```

We will see more of this in the upcoming chapters.

### Being deployment specific with instance folders

Flask provides yet another way of configuration where we can efficiently manage deployment-specific parts. Instance folders allow us to segregate deployment-specific files from our version-controlled application. We know that configuration files can be separate for different deployment environments such as development and production, but there are many more files such as database files, session files, cache files, and other runtime files. So, we can say that an instance folder is like a holder bin for these kinds of files.

### How to do it...

By default, the instance folder is picked up from the application automatically if we have a folder named `instance` in our application at the application level:

```
my_app/
  - app.py
  - instance/
    - config.cfg
```

We can also explicitly define the absolute path of the instance folder using the `instance_path` parameter on our application object:

```python
app = Flask(__name__, instance_path='/absolute/path/to/instance/folder')
```

To load the configuration file from the instance folder, we will use the `instance_relative_config` parameter on the application object:

```python
app = Flask(__name__, instance_relative_config=True)
```

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
Flask Configurations

This tells the application to load the configuration file from the instance folder. The following example shows how this will work:

```python
app = Flask(
    __name__, instance_path='path/to/instance/folder',
    instance_relative_config=True
)
app.config.from_pyfile('config.cfg', silent=True)
```

**How it works...**

In the preceding code, first, the instance folder is loaded from the given path, and then, the configuration file is loaded from the file named `config.cfg` in the given instance folder. Here, `silent=True` is optional and used to suppress the error in case `config.cfg` is not found in the instance folder. If `silent=True` is not given and the file is not found, then the application will fail, giving the following error:

```
IOError: [Errno 2] Unable to load configuration file (No such file or directory): '/absolute/path/to/config/file'
```

It might seem that loading the configuration from the instance folder using `instance_relative_config` is redundant work and can be moved to one of the configuration methods. However, the beauty of this process lies in the fact that the instance folder concept is completely independent of configuration, and `instance_relative_config` just compliments the configuration object.

**Composition of views and models**

As we go big, we might want to structure our application in a modular manner. We will do this by restructuring our Hello World application.

**How to do it...**

1. First, create a new folder in our application and move all our files inside this new folder.
2. Then, create `__init__.py` in our folders, which are to be used as modules.
3. After that, create a new file called `run.py` in the topmost folder. As the name implies, this file will be used to run the application.
4. Finally, create separate folders to act as modules.

For More Information:  
www.packtpub.com/web-development/flask-framework-cookbook
Refer to the following file structure for a better understanding:

```
flask_app/
  - run.py
  - my_app/
    - __init__.py
    - hello/
      - __init__.py
      - models.py
      - views.py
```

First, the `flask_app/run.py` file will look something like the following lines of code:

```python
from my_app import app
app.run(debug=True)
```

Then, the `flask_app/my_app/__init__.py` file will look something like the following lines of code:

```python
from flask import Flask
app = Flask(__name__)

import my_app.hello.views
```

Then, we will have an empty file just to make the enclosing folder a Python package, `flask_app/my_app/hello/__init__.py`:

```
# No content.
# We need this file just to make this folder a python module.
```

The models file, `flask_app/my_app/hello/models.py`, has a non-persistent key-value store:

```python
MESSAGES = {
    'default': 'Hello to the World of Flask!','
}
```

Finally, the following is the views file, `flask_app/my_app/hello/views.py`. Here, we fetch the message corresponding to the key that is asked for and also have a provision to create or update a message:

```python
from my_app import app
from my_app.hello.models import MESSAGES

@app.route('/')
@app.route('/hello')
def hello_world():
```
Flask Configurations

```python
return MESSAGES['default']
```

```python
@app.route('/show/<key>')
def get_message(key):
    return MESSAGES.get(key) or "%s not found!" % key
```

```python
@app.route('/add/<key>/<message>')
def add_or_update_message(key, message):
    MESSAGES[key] = message
    return "%s Added/Updated" % key
```

Remember that the preceding code is nowhere near production-ready. It is just for demonstration and to make things understandable for new users of Flask.

**How it works...**

We can see that we have a circular import between `my_app/__init__.py` and `my_app/hello/views.py`, where, in the former, we import `views` from the latter, and in the latter, we import the `app` from the former. So, this actually makes the two modules depend on each other, but here, it is actually fine as we won't be using `views` in `my_app/__init__.py`. We do the import of `views` at the bottom of the file so that they are not used anyway.

We have used a very simple non-persistent in-memory key-value store for the demonstration of the model layout structure. It is true that we could have written the dictionary for the `MESSAGES` hash map in `views.py` itself, but it's best practice to keep the model and view layers separate.

So, we can run this app using just `run.py`:

```
$ python run.py
* Running on http://127.0.0.1:5000/
* Restarting with reloader
```

The reloader indicates that the application is being run in the debug mode, and the application will reload whenever a change is made in the code.

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
Now, we can see that we have already defined a default message in `MESSAGES`. We can view this message by opening `http://127.0.0.1:5000/show/default`. To add a new message, we can type `http://127.0.0.1:5000/add/great/Flask%20is%20great!!`. This will update the `MESSAGES` key-value store to look like the following:

```python
MESSAGES = {
    'default': 'Hello to the World of Flask!',
    'great': 'Flask is great!!',
}
```

Now, if we open the link `http://127.0.0.1:5000/show/great` in a browser, we will see our message, which, otherwise, would have appeared as a not-found message.

See also

- The next recipe, *Creating a modular web app with blueprints*, provides a much better way of organizing your Flask applications and is a readymade solution to circular imports.

### Creating a modular web app with blueprints

A **blueprint** is a concept in Flask that helps make large applications really modular. They keep application dispatching simple by providing a central place to register all the components in the application. A blueprint looks like an application object but is not an application. It looks like a pluggable application or a smaller part of a bigger application, but it is not so. A blueprint is actually a set of operations that can be registered on an application and represents how to construct or build an application.

**Getting ready**

We will take the application from the previous recipe, *Composition of views and models*, as a reference and modify it to work using blueprints.

**How to do it...**

The following is an example of a simple Hello World application using blueprints. It will work in a manner similar to the previous recipe but is much more modular and extensible.

First, we will start with the `flask_app/my_app/__init__.py` file:

```python
from flask import Flask
from my_app.hello.views import hello

app = Flask(__name__)
app.register_blueprint(hello)
```

For More Information:

Flask Configurations

Next, the views file, my_app/hello/views.py, will look like the following lines of code:

```python
from flask import Blueprint
from my_app.hello.models import MESSAGES

hello = Blueprint('hello', __name__)

@hello.route('/
@hello.route('/hello')
def hello_world():
    return MESSAGES['default']

@hello.route('/show/<key>')
def get_message(key):
    return MESSAGES.get(key) or "%s not found!" % key

@hello.route('/add/<key>/<message>')
def add_or_update_message(key, message):
    MESSAGES[key] = message
    return "%s Added/Updated" % key
```

We have defined a blueprint in the flask_app/my_app/hello/views.py file. We don't need the application object anymore here, and our complete routing is defined on a blueprint named hello. Instead of @app.route, we used @hello.route. The same blueprint is imported in flask_app/my_app/__init__.py and registered on the application object.

We can create any number of blueprints in our application and do most of the activities that we would do with our application, such as providing different template paths or different static paths. We can even have different URL prefixes or subdomains for our blueprints.

How it works...

This application will work in exactly the same way as the last application. The only difference is in the way the code is organized.

See also

- The previous recipe, Composition of views and models, is useful to get a background on how this recipe is useful.

For More Information:
www.packtpub.com/web-development/flask-framework-cookbook
Making a Flask app installable using setuptools

So, we have a Flask application now, but how do we install it just like any Python package? It is possible that any other application depends on our application or our application is in fact an extension for Flask and would need to be installed in a Python environment so that it can be used by other applications.

How to do it...

Installing a Flask app can be achieved very easily using the setuptools library of Python. We will have to create a file called setup.py in our application's folder and configure it to run a setup script for our application. It will take care of any dependencies, descriptions, loading test packages, and so on.

The following is an example of a simple setup.py script for our Hello World application:

```python
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
import os
from setuptools import setup

setup(
    name = 'my_app',
    version='1.0',
    license='GNU General Public License v3',
    author='Shalabh Aggarwal',
    author_email='contact@shalabhaggarwal.com',
    description='Hello world application for Flask',
    packages=['my_app'],
    platforms='any',
    install_requires=[
        'flask',
    ],
    classifiers=[
        'Development Status :: 4 - Beta',
        'Environment :: Web Environment',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: GNU General Public License v3',
        'Operating System :: OS Independent',
    ]
)
```

For More Information:
www.packtpub.com/web-development/flask-framework-cookbook
How it works...

In the preceding script, most of the configuration is self-explanatory. The classifiers are used when we make this application available on PyPI. These will help other users search the application using these classifiers.

Now, we can just run this file with the install keyword as shown here:

```
$ python setup.py install
```

This will install this application along with all its dependencies mentioned in install_requires, that is, Flask and all the dependencies of Flask as well. Then, this app can be used just like any Python package in our Python environment.

See also

- The list of valid trove classifiers can be found at https://pypi.python.org/pypi?action=list_classifiers

For More Information:

www.packtpub.com/web-development/flask-framework-cookbook
Where to buy this book

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.