



INSTANT
Short | Fast | Focused

Flask Web Development

Tap into Flask to build a complete application in a style that you control

Ron DuPlain

[PACKT]
PUBLISHING

www.it-ebooks.info

Instant Flask Web Development

Tap into Flask to build a complete application in a style that you control

Ron DuPlain



BIRMINGHAM - MUMBAI

Instant Flask Web Development

Copyright © 2013 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: August 2013

Production Reference: 1230813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-78216-962-8

www.packtpub.com

Credits

Author

Ron DuPlain

Project Coordinator

Esha Thakker

Reviewers

Maxime Bouroumeau-Fuseau

Zach Denton

Proofreader

Clyde Jenkins

Acquisition Editor

Mary Nadar

Production Coordinator

Adonia Jones

Commissioning Editor

Mohammed Fahad

Cover Work

Adonia Jones

Technical Editor

Anita Nayak

Cover Image

Sheetal Aute

About the Author

Ron DuPlain is a core contributor to the Flask project, and has built dozens of production applications with Flask. He has been thinking in Python since 2005, applying it to a variety of applications from embedded systems to web development. He is based in Charlottesville, Virginia, USA, where he runs the local Python meetup group and is an organizer of the local bar camp un conferences. He holds a Bachelors in Science in Computer engineering from the University of Cincinnati and a Masters in Systems engineering from the University of Virginia.

I would like to thank Armin Ronacher for Flask and adding me to the project. I learned a lot from him. Thanks to Nicholas Skelsey and Matthew Sullivan for reviewing early manuscripts of this book. I dedicate this book to the Flask community, and all the programmers getting started with web development.

About the Reviewers

Maxime Bouroumeau-Fuseau is a Full Stack web developer with many years of experience building large web applications. Working for a fast growing startup allowed him to learn the intricacies of scaling to millions of users. He has been working with PHP for ten years but has been using Python for the last few years as his preferred language. He loves the world of web development and building beautiful products that reach as many people as possible and make their life easier.

Zach Denton is a CS major at Dartmouth College. He helps run the Dartmouth Hacker Club, a group of students that write programs for fun. They use Flask extensively in their projects. An avid hackathoner, Zach has over fifty repos on GitHub (<https://github.com/zacharydenton>) and can be found on the web at <http://zacharydenton.com/>.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.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.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Instant Flask Web Development	7
Preparing for development (Simple)	7
Running a simple application (Simple)	11
Routing URLs and accepting requests (Simple)	14
Handling requests and responses (Simple)	19
Handling static files (Simple)	23
Using a database (Simple)	25
Handling forms and file uploads (Simple)	32
Templating with Jinja – setting a base template (Simple)	36
Creating a new record (Intermediate)	39
Displaying a record (Intermediate)	42
Editing a record (Intermediate)	45
Listing all records (Simple)	47
Deleting a record (Advanced)	49
Using custom template filters in Jinja (Advanced)	50
Sending error responses (Simple)	52
Authenticating users (Advanced)	55
Handling sessions and users (Intermediate)	59
Deploying to the world (Advanced)	62

Preface

Flask started as a project to show you how you can build your own framework on existing Python toolkits. Actually, Flask started as an April Fool's joke in 2010. Several microframeworks had been appearing in the Python community, offering their own set of interfaces to build web services and applications entirely in one `.py` file. Having built Werkzeug (a web programming toolkit) and Jinja (a templating engine), Armin Ronacher zipped these two projects, put the zip file base64-encoded into a `.py` file, added a few small functions, and called it the **Denied** microframework.

The April 1st marketing site for Denied included a working "Hello World" application in a few lines of code, a spoof screencast, some bogus testimonials, zero documentation, and a few hints that you should not trust everything you read on the Internet. Denied was a critique on how an open source project can go wrong—mostly through missing documentation and design flaws that can cause serious issues in production programs.

This was Flask's beginning, the microframework with good intentions. Flask aimed to provide a good developer experience through clear, simple interfaces packaged with useful documentation, a thorough test suite, and careful thought to production issues and backward/forward-compatibility across releases. Flask demonstrated how to build a thin layer on top of Werkzeug and Jinja. Then Flask became popular on its own, developed a community, and now several Flask-built applications are in production, small and large.

As a contributor to the Flask project, I observed that most new users ask questions of the form, "what is the Flask way to do X?" Most of the time, the question is better stated as, "what is the Python way to do X?" Flask does not have many options about how you use it, which makes it a great tool to start simply and grow carefully without fighting the options, deeply embedded in the framework as your project gets large.

That said, when you are getting started with Flask, you will wonder how a representative application looks. This book will show you just that. I have condensed my own experience with Flask into a small but complete application which we will build bottom-up one piece at a time. Along the way, I will provide pointers as you make considerations in your own projects. If, by the end of this book, you want more, take a look at the scheduler application that we built, identify areas where you would like to design your own improvements, and use these areas for projects to exercise your web development skills.

What this book covers

Preparing for development (Simple) explains how to set up your development environment and ensure that you have everything in place for developing web projects with Python and Flask.

Running a simple application (Simple) shows how to warm up with the simplest "Hello World" Flask application.

Routing URLs and accepting requests (Simple) shows how to start your project by accepting web requests according to your URL rules.

Handling requests and responses (Simple) explains how Flask handles web requests and how you can build responses.

Handling static files (Simple) explains how to send files from disk as a response.

Using a database (Simple) explains how to declare a data structure, store data with your application, and manage queries with SQLAlchemy.

Handling forms and file uploads (Simple) explains how to build a form with WTForms to match the application data structure, which provides HTML input and validates user input.

Templating with Jinja – setting a base template (Simple) explains how to set the base HTML structure of your application, providing page structure and theme able CSS style with Twitter Bootstrap and JavaScript page manipulation with jQuery.

Creating a new record (Intermediate) explains how to provide a web form to create a database record.

Displaying a record (Intermediate) explains how to retrieve a record from the database and display it in HTML.

Editing a record (Intermediate) explains how to provide a web form to edit an existing database record.

Listing all records (Simple) explains how to build an index view to display all records from the database in HTML.

Deleting a record (Advanced) explains how to add a JavaScript-triggered `DELETE` request to remove a record from the database.

Using custom template filters in Jinja (Advanced) explains how to extend Jinja to include custom display formats for your data.

Sending error responses (Simple) explains how to respond to errors with styled HTML pages.

Authenticating users (Advanced) explains how to build a user data model which includes a hashed password field for authentication by credential that only the user would know.

Handling sessions and users (Intermediate) explains how to keep users logged in for on-going requests after authentication.

Deploying to the world (Advanced) explains how to publish your application with nginx and gunicorn on Ubuntu, with a pointer on how to deploy on any operating system using HTTP proxying.

What you need for this book

This book assumes that you have Python and an interest in web development. You need to know, or be willing to learn some Python and a little bit of the web technologies HTTP, HTML, CSS, and JavaScript. This book specifically covers web development with Flask, which is a third-party package for the Python programming language. If you are new to Python or web development, be sure to have good resources on hand for the Python language and the various web technologies.

At the time of this writing, Flask is version 0.10, which is stable and production-ready. From its beginning, the Flask project has made every effort to maintain backward compatibility, providing upgrade instructions on major releases. See <https://pypi.python.org/pypi/Flask> for the latest version and installation instructions for Flask.

This book uses **virtualenv** to install third-party packages, which is version 1.9 at the time of this writing. Note that virtualenv 1.9 adds SSL support when installing packages, so if you already have virtualenv installed, be sure to upgrade. See <https://pypi.python.org/pypi/virtualenv> for the latest version and instructions for virtualenv.



This book uses both Python 2.7 and Python 3.3.

At the time of this writing, Flask supports Python 3.3+ as of its 0.10 release. Packages available in the community, however, do not always have Python 3 support. I encourage you to use Python 3 in your projects. The Python community is still in transition to Python 3, which is backward-incompatible with Python 2, but investing in Python 3 now will allow you to grow your codebase with modern Python, where new features are being added to the language.

If you are using a Python implementation other than **CPython**, such as **pypy**, download the Flask source and run its tests to verify that Flask is compatible with your interpreter.

Other materials are listed in the *Checklist for Development Environment* section under the *Preparing for development* recipe.

Who this book is for

This book is for readers who are new to web programming, or are familiar with web programming but new to Flask. This book is intended for three classes of readers:

- ▶ Programmers getting started in web development, starting with an interest in Python towards a working knowledge of how to build web applications
- ▶ Programmers who are familiar with web development, starting with a working knowledge of how to build applications in a familiar language (which may or may not be Python) towards the use of Flask in daily work
- ▶ Designers who have decided to learn programming, starting with a user interface design in mind toward a self-built functional application

In all the three cases, Flask is well suited as a starting place for web development in Python, and Python as a language is accessible to readers who are beginning to learn programming.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "This creates a Python object `app`, which is a WSGI application."

A block of code is set as follows:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, world!'

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 import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, world!'

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

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

```
$ python manage.py runserver -t 0.0.0.0 -p 8080
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "We reuse the same template as the create form, but we say **Edit** instead of **Add**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really find useful.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic in which you have expertise, and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Instant Flask Web Development

Welcome to *Instant Flask Web Development*. We will progressively walk through web development in the Python programming language using Flask, a small but expressive framework which provides the essentials and enables you to build your own code patterns. We will build a simple scheduling application to keep track of appointments, including a database and a user interface, and we will build one piece at a time. We will build our application bottom-up, and you will see how everything fits together in later sections. This bottom-up approach will give you the building blocks you need to grow your project with Flask.

Preparing for development (Simple)

We begin our exploration of Flask web programming with everything in its place, setting up a development environment in Python.

Getting ready

Go to `python.org` to get Python. Our application will run on Python 2.7 and Python 3.3+. Once you have Python, you do not need administrative access to your development machine, and you can even install Python for just your user according to the install instructions on `python.org`.

How to do it...

Python projects can manage packages using **virtualenv**, a user-writable area on the machine that is dedicated to a specific project. You can use virtualenv on Unix/Unix-like systems, Mac OS X, and Windows.

We'll use the command line to get virtualenv up and running, then discuss the tools you need in your development environment. Before we get started, note the *Common Errors and how to solve them* subsection in the *There's more...* section in case you run into a problem.

Unix-style systems ship with a terminal emulator, which we will use throughout this book. The steps required to install Flask on Unix, Unix-like systems, and Mac OS X are as follows:

1. Open a terminal in the directory where you would like your project to be located.
2. Download virtualenv from pypi.python.org/pypi/virtualenv/1.9.1. In the terminal, you can do this with `curl -O https://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.9.1.tar.gz`.
3. Unpack virtualenv with `tar xvzf virtualenv-1.9.1.tar.gz`.
4. Create a virtualenv tool named `env` with `python virtualenv-1.9.1/virtualenv.py env`. You can use any name you like; just be sure to make the changes in the commands here according to the name you choose.
5. Activate the virtualenv tool with `. env/bin/activate`.
6. Install Flask with `pip install Flask`.
7. Verify whether Flask is installed with a simple smoke test, `python -m flask.config`. Nothing will be displayed if Flask is installed.

You can turn off the virtualenv with `deactivate` to continue using the terminal for other projects. Anytime you resume work on your Flask project, activate the virtualenv again with `. path/to/env/bin/activate`. The `.` command sources the activate script to set environment variables that point to the `python` executable in the virtualenv and enable import of the packages installed there.

For Windows, we will use `cmd.exe`. The steps to install Flask on Windows are as follows:

1. Ensure the Python installation and its `scripts` directory, which are `C:\Python33\` and `C:\Python33\Scripts\` by default, are in your current `PATH` variable. Use `Python27` if you are using Python 2.7, and note that Windows uses a semicolon in-between directories in `PATH`. Use `echo %PATH%` to see your `PATH` variable. In general, edit `PATH` by navigating to **Computer | Properties | Advanced | Environment Variables**.
2. Open a shell in the directory where you would like your project to be located. Historically, you could run into scripts which fail on spaces in the filepath, which is why `C:\Python27\` is preferred to `C:\Program Files\Python27\`; keep that in mind if you see strange errors.
3. Download `virtualenv.py` from <https://raw.githubusercontent.com/pypa/virtualenv/1.9.1/virtualenv.py> to the same directory.
4. Create a virtualenv named `env` with `python virtualenv.py env`. You can use any name you like; just be sure to make the changes in the commands here according to the name you chose.

5. Activate the virtualenv with `env\Scripts\activate.bat`.
6. Install Flask with `pip install Flask`.
7. Verify whether Flask is installed with a simple smoke test, `python -m flask.config`. Nothing will be displayed if Flask is installed.

You can turn off the virtualenv tool with `deactivate` to continue using the command prompt for other projects. Anytime you resume work on your Flask project, activate the virtualenv tool again with `env\Scripts\activate.bat`, which is a batch file to set environment variables that point to the python executable in the virtualenv and enable import of the packages installed there.



If you are using Windows, you can use PowerShell for a richer command-line environment by referring to technet.microsoft.com/en-us/library/bb978526.aspx, or you can work with `cmd.exe` if you do not have access to PowerShell by referring to docs.python.org/2/using/windows.html.

How it works...

Python itself ships with a large collection of utilities in its standard library. With Python out of the box, you can immediately use any of the code listed at docs.python.org/2/library/. The Python interpreter has several functions and objects which are always available, and several more which become available with an import statement as shown:

```
import antigravity
```

The Python community adds even more modules to import through collaborations on frameworks (including Flask) and toolkits published to the Python Package Index, PyPI (pronounced "pie p.i."), at pypi.python.org. Packaging gets complicated, and pip and virtualenv aim to make a manageable workflow around use of third-party packages on PyPI.

There's more...

There's a lot more that can be achieved as follows:

Checklist for the development environment

Here are the items you need to develop a web application:

- ▶ A text editor for adding and editing source code. I use emacs and vim, which have highly productive keyboard shortcuts, and Sublime Text is a popular choice. There are many options; just be sure the editor is designed for code.
- ▶ Python with a virtualenv tool, using the instructions in this chapter.

- ▶ A terminal console to run your Flask code, one for each Flask application you run, as discussed in the next section.
- ▶ A terminal console to run Python's interactive interpreter. Here you can poke at APIs, experiment with code, verify unit functionality, and use Python's built-in `help(arg)` function to get online documentation.
- ▶ A modern web browser. Firefox and Chrome have readily accessible JavaScript consoles and document inspectors for web pages. The JavaScript console and document inspector are essential for web user interface development.
- ▶ Flask docs available at: flask.pocoo.org/docs/.
- ▶ Werkzeug docs available at: werkzeug.pocoo.org/docs/. Werkzeug is Flask's underlying web service implementation.
- ▶ Jinja docs available at: jinja.pocoo.org/docs/. Jinja is Flask's default template engine.
- ▶ Python docs available at: docs.python.org/2/.
- ▶ Version control system. Track changes to your project. I personally recommend git, available at git-scm.com.

An integrated development environment (IDE) can bundle these tools together in one application, and there are several available for Python. Personally, I find the best tool for each item in the checklist and put together my own environment as a collection of standalone tools and applications. I highly recommend having a stand-alone command-line available for code interaction, so that you can always get to the core of your code even if you have an IDE that you like.

(I find myself to be more productive doing everything on the command-line and in the terminal, but that's a matter of personal preference. Be sure to love your development environment, so that you can think clearly about your work.)

Common errors and how to solve them

Two common errors you will see in using a Python virtualenv are:

- ▶ The `python` command is not found when you attempt to run Python
- ▶ "No module named flask" when you attempt to import Flask in your code

Both issues are caused by an incorrect environment variable named **PATH** in your command-line session, which is an ordered list of directories your system should search when loading a command. If `python` is not found, your **PATH** was not updated when you installed Python and you should revisit the Python installer. If `flask` is not found, you have either not activated your virtualenv or you have not installed Flask, and you should revisit the install instructions here.

Running a simple application (Simple)

Let's get started by running a simplest Flask application.

Getting ready

Make sure you have activated your virtualenv tool as described in the previous section.

How to do it...

1. Put this code into a file named `hello.py`:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, world!'

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

2. Run this application from the console:

```
$ python hello.py
```

3. If this runs correctly, you will see:

* Running on `http://127.0.0.1:5000/`

This is a URL, which you can visit in your browser. When you do, your browser will display a plain page which says **"Hello, world!"** (without quotes). Note that the address `127.0.0.1` refers to localhost, such that Flask will only respond to requests from the host running the code.

How it works...

Line-by-line, here is what this code does:

```
app = Flask(__name__)
```

This creates a Python object `app`, which is a **WSGI** application. WSGI stands for **Web Service Gateway Interface**, and is the Python community's standard interface for hosting web projects. Any WSGI resource or tool that you find, you can apply to this `app` object, including WSGI middleware (if you do, wrap `app.wsgi_app`) and HTTP servers such as gunicorn or Apache `httpd` with `mod_wsgi`.

The argument given to the `Flask` class tells Flask how to find resources associated with your application: static files and templates. Passing `__name__` tells Flask to look at the current Python module—the `.py` file containing this code. Most of your applications will use `__name__`; the Flask docs describe this parameter in detail if you ever suspect `__name__` is not fulfilling your needs.

The next block sets up a function to handle web requests to the `'/'` URL:

```
@app.route('/')
def hello():
    return 'Hello, world!'
```

Every time Flask gets a request to the `'/'` URL, it will call the `hello` function. The Python web community calls these routed functions **view** functions. In general, view functions in Flask return strings for web responses, and here we provide a simple "Hello, world!" response for all the requests, to prove that things are working.

The final block tells Python to run a development web server, but to only do so if the current `.py` file is being called directly:

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

That is, this code block will run if you run the command `python hello.py` but not if you use `import hello` from another Python module.

There's more...

Once you are satisfied with the Hello World application, let's structure our project.

Project layout

We will be building a simple scheduling application to manage appointments and display them. Let's move our one-file application into a directory setup for a larger application. Create the following file layout in your project, with a `sched` directory which contains subdirectories `static` and `templates`, both currently empty.

```
manage.py
requirements.txt
sched/
```

```
app.py
config.py
filters.py
forms.py
__init__.py
models.py
static/
templates/
```

Move `hello.py` to `app.py` inside the `sched` directory. The `__init__.py` file is an empty file telling Python that `sched` is a package containing Python files. The `config.py`, `filters.py`, `forms.py`, and `models.py` files are currently empty files which we will fill in the upcoming sections.

Development server

If you need to access your Flask application from another machine on the network, use:

```
app.run('0.0.0.0')
```

The default port for Flask's development server is 5000, and we will use `localhost` on port 5000 throughout this book. If this is already in use, or you would like to use a different port, you can set it via the second argument to the `run` method, as follows.

```
app.run('0.0.0.0', 8080)
```

The development server given by `app.run` is for development only. When you are ready to publish your application, use an industrial strength web server as discussed in the later section on deploying and logging. Because we are using `app.run` for development only, let's turn on the debug mode, which will provide an interactive debugger in the web browser when uncaught exceptions occur and will reload code on changes to existing Python files in your project. You should only use `'0.0.0.0'` and/or debug mode on a trusted network. Add `debug=True` to the `run` line, keeping any other arguments you already added:

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

Command-line interface with Flask-Script

We will use `manage.py` to manage our application using **Flask-Script**, with these contents:

```
from flask.ext.script import Manager
from sched.app import app

manager = Manager(app)
app.config['DEBUG'] = True # Ensure debugger will load.

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

The `requirements.txt` file is a conventional way to track all third-party Python packages. Set the file with the following listing, which is one package name per line. Run `pip install -r requirements.txt` with your virtualenv active and a working internet connection to install these dependencies.

```
Flask
Flask-Script
```

Now you can run your application with `python manage.py runserver`. Run `python manage.py -h` to get help and `python manage.py runserver -h` to get help on the development server options. Use `python manage.py shell` to get an interactive Python interpreter with the Flask application loaded, which you can use to inspect your code and try new things interactively. By default, Flask-Script gives you the `runserver` and `shell` commands; you can add custom `manage.py` subcommands; refer to the Flask-Script docs.

As we discussed, you can make the development server accessible from another machine on the network and change the port; here is the command pattern:

```
$ python manage.py runserver -t 0.0.0.0 -p 8080
```



Flask-Script added Python 3 support earlier this month. This book now fully supports Python 2 and Python 3.

Routing URLs and accepting requests (Simple)

This section will connect URLs to Python functions in our web service.

Getting ready

Use the project layout listed under the *Project Layout* section in the *Running a Simple Application* recipe.

How to do it...

1. The Flask app object includes a `route` decorator to specify URL rules to use for a view function conveniently, which provides a declarative style for routing requests to Python callables. The following code routes five URL handling functions for the list, detail, creation, updating, and deletion of appointment records in our scheduling application. For now, we simply return a string describing what the handler is going to do, which we'll implement later, so that you can see the URL routing in action from your browser. Place into `app.py`:

```
@app.route('/appointments/')
def appointment_list():
    return 'Listing of all appointments we have.'

@app.route('/appointments/<int:appointment_id>/')
def appointment_detail(appointment_id):
    return 'Detail of appointment      #{}.format(appointment_id)

@app.route(
    '/appointments/<int:appointment_id>/edit/',
    methods=['GET', 'POST'])

@app.route(...) and def appointment_edit(...).

def appointment_edit(appointment_id):
    return 'Form to edit appointment #.'.format(appointment_id)

@app.route(
    '/appointments/create/',
    methods=['GET', 'POST'])
def appointment_create():
    return 'Form to create a new appointment.'

@app.route(
    '/appointments/<int:appointment_id>/delete/',
    methods=['DELETE'])
def appointment_delete(appointment_id):
    raise NotImplementedError('DELETE')
```


Downloading the example code

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

2. Using `python manage.py runserver`, you can visit these URLs at:

- ❑ `http://localhost:5000/appointments/`
- ❑ `http://localhost:5000/appointments/1/`
- ❑ `http://localhost:5000/appointments/1/edit/`
- ❑ `http://localhost:5000/appointments/create/`
- ❑ `http://localhost:5000/appointments/1/delete/`



If you test these URLs in your browser, you will find that the delete URL responds with a **405 Method Not Allowed** error. This is intentional, because the browser sends a `GET` request by default and we are only allowing `DELETE` methods. We do not want to delete a record on a `GET` request, but only when our delete button is pressed (built in a later section).

3. You can build URLs to your view functions using the `flask.url_for` function, which returns a string representation of the URL. This lets you program using clean identifiers in case URLs change, using the names of the target function and its arguments:

```
from flask import url_for

@app.route('/appointments/<int:appointment_id>/')
def appointment_detail(appointment_id):
    edit_url = url_for('appointment_edit',
        appointment_id=appointment_id)
    # Return the URL string just for demonstration.
    return edit_url
```

4. The first argument to `url_for` is called the endpoint, which by default is the name of the Python function wrapped by the `app.route` decorator. You can override the default using the `endpoint` keyword argument to `app.route`:

```
@app.route(
    '/appointments/<int:appointment_id>/',
    endpoint='some_name')
```

```
def appointment_detail(appointment_id):  
    # Use url_for('some_name', appointment_id=x)  
    # to build a URL for this.  
    return 'Just to demonstrate...'
```

How it works...

The main argument to `app.route` is the string URL rule as implemented by **Werkzeug**, Flask's underlying toolkit for all things WSGI. Items listed in angle brackets `<argument>` are parsed as named arguments to pass into the view function. Flask uses a convention of `<converter:argument>` in the URL rule to parse the argument value before passing it to your view function, and only routing the URL if a value is correctly parsed. By default, Flask treats the argument as a string. The additional built-in converters are:

- ▶ `int`: The value of this converter is an integer
- ▶ `float`: The value of this converter is a floating point number
- ▶ `path`: The value of this converter is a string such as the default, but also accepts slashes

You can define your own converters as shown in Flask's `app.url_map` documentation, but you may not need to, such that your view function can inspect the string argument and parse what it needs.

By default, if a URL rule ends in a trailing slash `/`, Flask will redirect requests without the trailing slash to the handler which includes it.

There's more...

Once you are up and routing, there are a few things you should know about HTTP and Flask routes.

Handling HTTP methods

The most common keyword argument to `app.route` is `methods`, giving Flask a list of HTTP methods to accept when routing, which when absent defaults to `GET`. Valid values are `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, and `OPTIONS`. RFC2068 is a standards document which defines these methods. Briefly, they are:

- ▶ `GET`: This option is used to reply with information on resource, most common
- ▶ `POST`: This option is used to receive from browser/client updated information for resource
- ▶ `PUT`: This option is like `POST`, but repeat `PUT` calls on a resource should have no effect
- ▶ `DELETE`: Using this option removes the resource

- ▶ HEAD: This option is like GET, but replies only with HTTP headers and not content
- ▶ OPTIONS: This option is used to determine which methods are available for resource



Flask implements HEAD for you if GET is present, and OPTIONS for you in all cases. Old implementations of HTTP browsers only supported GET and POST, but we will add a DELETE request by JavaScript in a later section. It is up to you and your project to use the other HTTP methods, particularly in cases where the semantics of the methods are important.

An alternative to decorating functions

The `@app.route` approach is a Python decorator, which you use at the time you define a function. As an alternative, you can use `app.add_url_rule`, which works exactly like `app.route` but is a simple method call and is not a decorator. You can provide `app.add_url_rule` with any Python callable, accepting the parameters in the URL rule, which will become `request.view_args` at the time of the request. If you have your own ideas on how to specify the URL routes of your application, you can use whatever tools you like with `app.add_url_rule` as a utility to wire up your Flask application.

Route collisions

When Flask routes your requests to unexpected places, look for collisions in your `app.route` calls. That is, if you have routes `/<path:foopath>` and `/foo/bar/baz/`, both will match on `/foo/bar/baz/`. The solution is to be as specific as possible in your route parameters, and avoid overly generic parameters.

Routing with subdomains

You can route subdomains using the `subdomain` keyword argument to `app.route`:

```
@app.route('/', subdomain='<spam_eggs>')
def subdomain_example(spam_eggs):
    return '...'
```

This subdomain argument uses the same parameter approach as other URLs, but note that proper subdomains are limited in what they can accept. Simple names are straightforward, but if you have specific requirements, see RFC2181 for name syntax and note that some HTTP clients do not support the full spec in subdomains.

When you use subdomains, Flask needs to know the server name in order to parse the subdomain from the URL. Provide `SERVER_NAME` to `app.config`, as described in the section on configuration.

You will also run into limitations while developing locally on your machine, because `localhost` does not accept subdomains. You can set your operating system's hosts file (typically `/etc/hosts`). If your `SERVER_NAME` is `example.com` and your subdomain argument is `foo`, you can set:

```
127.0.0.1    localhost local.example.com foo.local.example.com
```

The `hosts` file does not accept wildcards. If you have access to a domain name service, an alternative to hosts file is to set an A record and all of its subdomains (wildcard) in DNS to `127.0.0.1`, like the following example. This will route all subdomains for `local.example.com` on all machines to the local host.

```
local.example.com A 127.0.0.1
*.local.example.com A 127.0.0.1
```

This technique requires a DNS network call, so if you are working offline, you'll need to fall back on the hosts file.

Handling requests and responses (Simple)

This section will demonstrate how Flask handles incoming data on HTTP requests and how you can send responses.

Getting ready

Set aside the scheduling application and open a new Python file to explore Flask.

How to do it...

1. At the core, Flask and Werkzeug provide request and response objects to represent incoming and outgoing data for your web application. Flask provides three different patterns for return values from your view functions:
 - ❑ string, which can optionally use a template engine (introduced later)
 - ❑ a response instance, an object with attributes representing HTTP response details
 - ❑ tuple of `(string, status)` or `(string, status, http_headers)`, for convenience, such that you do not have to create a response instance

2. Let's see each of the response patterns in action. Each response will say **Hello, world!** with a 200 OK status code (the typical HTTP response on success), and the latter two functions tell the browser to display the response as plain text. The first function's response displays as though it were HTML because a string-only return object has no means to tell Flask how to set the content-type of the response.

```
from flask import Flask, make_response

app = Flask(__name__)

@app.route('/string/')
def return_string():
    return 'Hello, world!'

@app.route('/object/')
def return_object():
    headers = {'Content-Type': 'text/plain'}
    return make_response('Hello, world!', status=200,
                        headers=headers)

@app.route('/tuple/')
def return_tuple():
    return 'Hello, world!', 200, {'Content-Type':
                                'text/plain'}
```

3. As Flask prepares to call your view function and accepts its return value, it walks through each of the before-request and after-request callbacks you provide. Here, we set up a before-request function and an after-request function, purely for demonstration purposes to illustrate Flask's request and response handling. You can use these handlers to explore Flask behavior when you want to interact with what Flask docs are saying. When you run this code, keep in mind that browsers often automatically look for `/favicon.ico`, which can cause extra requests for your application when testing.

```
from flask import request

def dump_request_detail(request):
    request_detail = ""
    # Before Request #
    request.endpoint: {request.endpoint}
    request.method: {request.method}
    request.view_args: {request.view_args}
    request.args: {request.args}
    request.form: {request.form}
    request.user_agent: {request.user_agent}
    request.files: {request.files}
```

```

request.is_xhr: {request.is_xhr}

## request.headers ##
{request.headers}
    """.format(request=request).strip()
    return request_detail

@app.before_request
def callme_before_every_request():
    # Demo only: the before_request hook.
    app.logger.debug(dump_request_detail(request))

@app.after_request
def callme_after_every_response(response):
    # Demo only: the after_request hook.
    app.logger.debug('# After Request #\n' + repr(response))
    return response

```

How it works...

Here are commonly used features of the request object. See the Flask docs for the full list of incoming request data. To provide an example, each description includes an example value for a request hitting `/string/?foo=bar&foo=baz` from our browser.

- ▶ **endpoint:** This feature of the request object specifies the name of the request endpoint routed, for example, `return_string`.
- ▶ **method:** This feature of the request object specifies the HTTP method of the current request, for example, `GET`.
- ▶ **view_args:** This feature of the request object specifies the dict of view function arguments parsed from URL route rule, for example, `{}`.
- ▶ **args:** This feature of the request object specifies the dict of arguments parsed from the URL query string, for example, `request.args['foo']` is `'bar'` and `request.args.getlist('foo')` is `['bar', 'baz']`.
- ▶ **form:** This feature of the request object specifies the dict of form data from POST or PUT requests, for example, `{}`.
- ▶ **user_agent:** This feature of the request object specifies the version identification provided by the browser.
- ▶ **files:** This feature of the request object specifies the dict of file uploads from POST or PUT requests, which go here instead of `request.form`, for example, `{}`. Each value in the dict is a `FileStorage` object which behaves like a Python file object, but also includes a `save(filepath)` method to store uploaded files (after you validate the destination path).

- ▶ `is_xhr: True`: This feature of the request object specifies when the incoming request is a JavaScript XMLHttpRequest, and `False` otherwise. This works with JavaScript libraries that provide the X-Requested-With HTTP header, set to XMLHttpRequest.

Flask uses a custom dict type `ImmutableMultiDict` which supports multiple values per key (accessed by the `getlist` method), in cases where HTTP allows multiple values for a given argument, such as the query string in the URL, for example, `?foo=bar&foo=baz`.

There's more...

We used `app.before_request` and `app.after_request` in our demonstration code. Eventually, you will want objects to stick around in your code between the before- and after-request handlers.

Before and after a request

If you need to keep an object around between the before and after request hooks, you can set an attribute on the `flask.g` object. The `g` object only lives for the duration of the request, and you can set anything you want on the `g` object. This is for convenience, for the things that you need throughout the request, but do not yet have a home. Do not abuse the `g` object by giving it objects that belong elsewhere, such as a database system.

Here is the `g` object in action, where a random integer between 0 and 9 is set on `g.x` and logged before and after the request:

```
import random

from flask import g

@app.before_request
def set_on_g_object():
    x = random.randint(0, 9)
    app.logger.debug('before request: g.x is {x}'.format(x=x))
    g.x = x

@app.after_request
def get_on_g_object(response):
    app.logger.debug(
        'after request: g.x is {g.x}'.format(g=g))
    return response
```

Handling static files (Simple)

Flask is ready to serve files on your disk from the moment you serve your first request.

Getting ready

Go to your scheduler project and look at the `static` folder inside the `sched` directory.

How to do it...

1. Put the files inside the `static` folder.
2. Build URLs for them using `flask.url_for('static', filename='path/to/filename')` where `path/to/filename` is the file path inside the static folder, using `/` regardless of the operating system you are using.

How it works...

By convention, Flask looks for a folder named `static` next to your application, and serves the files there at the `/static/<path:filename>` URL, matching all the files in your `static` folder and its subdirectories. That is, if your application is at `app.py`, then by default, Flask will look at the `static` folder next to `app.py`. Placing a file at `/static/img/favicon.ico` next to `app.py` becomes available at the URL `/static/img/favicon.ico`, which is `http://localhost:5000/static/img/favicon.ico` in the default development server. (This favicon is wired up in the next section on templating.)

There's more...

You can customize how Flask handles static files.

Serving static files in production

When you deploy your application, you will likely want to serve static files from an industrial strength HTTP server such as `nginx` or `Apache httpd`, which are highly optimized for static files. You can configure these servers to serve the `/static/` URL from the same `/static/` folder in your project. If you do not have access to these servers, you can use Flask's static file handling in production if you need to.

Hosting static files elsewhere

If you are using static files served by another server at a URL other than `/static/`, you can add routes to your Flask application with the `build_only` option. This tells Flask how to build URLs with the `flask.url_for` function, without needing a view function or having Flask attempt to serve those files.

```
app.add_url_rule('/attachments/<path:filename>',
                 endpoint='attachments', build_only=True)
```

Custom static file handler

You can build your own static file handler if you need to do any custom handling when serving files from the filesystem (for example, only serving a file to users who pay), for use in static files outside your static folder (for example, `/robots.txt`), or to override Flask's built-in static view function:

```
import os.path
from flask import Flask, send_from_directory

# Setting static_folder=None disables built-in static handler.
app = Flask(__name__, static_folder=None)
# Use folder called 'assets' sitting next to app.py module.
assets_folder = os.path.join(app.root_path, 'assets')

@app.route('/assets/<path:filename>')
def assets(filename):
    # Add custom handling here.
    # Send a file download response.
    return send_from_directory(assets_folder, filename)
```

Using a hard-coded directory and `send_from_directory` will mitigate directory traversal attacks, because the underlying implementation uses `flask.safe_join(directory, filename)` to sanitize the input, which you can use in your own code when handling filenames passed in as user input.

Use a custom static view function to do custom handling of static files, and not to change configuration of the `static` folder. If you just need to rename the `static` folder, use:

```
app = Flask(__name__, static_folder='foldername')
```

By default, Flask serves static files at `/static/` where `static` is the name of the folder provided in the `static_folder` argument, with default `static`. If you want a different URL path from the folder name, use:

```
app = Flask(__name__, static_url_path='/assets')
```

HTTP caching

By default, Flask sends an HTTP header with the static file telling the browser to cache the file for 43200 seconds, or 12 hours. You can configure this using the `SEND_FILE_MAX_AGE_DEFAULT` configuration variable. For example, if you want aggressive caching on all static files, you can set this value to 2592000 seconds, or 30 days:

```
app.config['SEND_FILE_MAX_AGE_DEFAULT'] = 2592000
```

Be careful when increasing the HTTP cache length, because browsers will have stale files when you deploy changes. To work around that issue, change the static URL path on redeploy:

```
app = Flask(__name__, static_url_path='/static/v2')
```

Directory index

Flask does not list the contents of the files in a static directory for user browsing; you have to link to the static files directly. If you need a directory index view, consider using an industrial strength static file web server such as nginx or Apache httpd.

Using a database (Simple)

Our scheduler application needs data, and we want to store that data properly so that we can stop and start our application without losing our appointments.

Getting ready

We are working from the `models.py` file inside the `sched` directory of our project.

How to do it...

SQLAlchemy provides a Python toolkit and object-relational manager for relational databases which use SQL. SQLAlchemy stands on its own, regardless of what web framework you use, and we can integrate it into Flask with Flask-SQLAlchemy, which will manage database connections with the Flask request lifecycle.

Update `requirements.txt`:

```
Flask
Flask-Script
Flask-SQLAlchemy
```

Then:

```
$ pip install -r requirements.txt
```

Let's model our appointment data structure using SQLAlchemy's declarative extension, which allows us to write Python classes to represent database tables. Note that Flask-SQLAlchemy includes some declarative functionality, but our application uses pure SQLAlchemy to make the code portable to any Python project and to let you reference the core SQLAlchemy docs, docs.sqlalchemy.org.

We are modeling an Appointment class which maps objects to an appointment table. Start with a Base class and define an Appointment subclass. SQLAlchemy will find all subclasses of Base when working with the underlying database system. We then define columns on the Appointment class which will map attributes on every Appointment object that we use in our code, storing and retrieving values as columns in the appointment table in our database. In `models.py`:

```
from datetime import datetime

from sqlalchemy import Boolean, Column
from sqlalchemy import DateTime, Integer, String, Text
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()

class Appointment(Base):
    """An appointment on the calendar."""
    __tablename__ = 'appointment'

    id = Column(Integer, primary_key=True)
    created = Column(DateTime, default=datetime.now)
    modified = Column(DateTime, default=datetime.now,
                      onupdate=datetime.now)

    title = Column(String(255))
    start = Column(DateTime, nullable=False)
    end = Column(DateTime, nullable=False)
    allday = Column(Boolean, default=False)
    location = Column(String(255))
    description = Column(Text)
```

To load our domain model in our application, we configure our database on our Flask app and setup a db object to get ready for queries in our request handlers, in `app.py`:

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

from sched.models import Base
```

```

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

# Use Flask-SQLAlchemy for its engine and session
# configuration. Load the extension, giving it the app object,
# and override its default Model class with the pure
# SQLAlchemy declarative Base class.
db = SQLAlchemy(app)
db.Model = Base

```

We can add a couple of helpers as methods on the Appointment class. A duration property provides a calculation of the length of the appointment, in seconds, using the start and end times of the appointment object. The `__repr__` method tells Python how to represent the appointment object when printing it. This implementation will say `<Appointment: 1>` instead of Python's default form of `<__main__.Appointment object at 0x26cf2d0>`.

Back in `models.py`, in the Appointment class declaration, add:

```

@property
def duration(self):
    delta = self.end - self.start
    return delta.days * 24 * 60 * 60 + delta.seconds

def __repr__(self):
    return (u'<{self.__class__.__name__}: {self.id}>'
            .format(self=self))

```

How it works...

Each Column takes a type, which gives structure to the appointment record. We give keyword arguments to Column to define behavior as follows:

- ▶ `primary_key`: True means that this field is the record identifier.
- ▶ `default`: When no data is given, use this value. This can be a function which returns a value, for example, set `created` to the output of `datetime.now()` to provide the date/time at the time the database record is created.
- ▶ `onupdate`: When a record is stored or updated, set its value to the return value of the given function, for example, set `modified` to the current date/time at the time the database record is updated.
- ▶ `nullable`: When False, do not allow records to be stored which do not have a value set for this attribute, by raising an exception.

The URL `sqlite:///sched.db` tells SQLAlchemy to use a SQLite database in the current working directory. Python ships with support for SQLite, which is an embedded relational database management system also available at, sqlite.org.

There's more...

We have only modeled our data and told Flask how to connect to a SQLite database. Let's try out some queries.

Querying with SQLAlchemy

With an Appointment model definition in place, we can run some queries from Python. We can make some sample queries before building out our application's view functions. Add the following code to the main script of `models.py`, inside an `if __name__ == '__main__':` block. Add any statements or `print` calls that you want, in order to see how SQLAlchemy works and watch it run with:

```
$ python models.py
```



Where is the web request? This section works with SQLAlchemy directly, outside a request context, in order to illustrate how SQLAlchemy works. This bottom-up approach helps you understand your building blocks.

We start with an engine, which connects to the database and executes queries. If you would rather not create a file on disk, you can use a temporary in-memory database. The `sqlite://` URL tells SQLAlchemy to connect to a SQLite database, and because the filepath is omitted, it should connect to a temporary database in memory. That is, the `sqlite://` URL would provide a database which only exists for the duration of Python's process execution, and will not persist across calls to Python.

```
from datetime import timedelta

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///sched.db', echo=True)
```

Next we create a session and create the database tables. When the engine connects to the database and executes queries, the session represents an on-going conversation with the database and is the primary entry point for applications to use a relational database in SQLAlchemy.

```
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()
```

Now we add some sample data. We generate times for our fake appointments using the current time now plus or minus some `timedelta`, which accepts days and seconds.

```
now = datetime.now()

session.add(Appointment(
    title='Important Meeting',
    start=now + timedelta(days=3),
    end=now + timedelta(days=3, seconds=3600),
    allday=False,
    location='The Office'))
session.commit()

session.add(Appointment(
    title='Past Meeting',
    start=now - timedelta(days=3),
    end=now - timedelta(days=3, seconds=3600),
    allday=False,
    location='The Office'))
session.commit()

session.add(Appointment(
    title='Follow Up',
    start=now + timedelta(days=4),
    end=now + timedelta(days=4, seconds=3600),
    allday=False,
    location='The Office'))
session.commit()

session.add(Appointment(
    title='Day Off',
    start=now + timedelta(days=5),
    end=now + timedelta(days=5),
    allday=True))
session.commit()
```

To create, update, and delete an appointment record:

```
# Create. Add a new model instance to the session.
appt = Appointment(
    title='My Appointment',
```

```
start=now,
end=now + timedelta(seconds=1800),
allday=False)

session.add(appt)
session.commit()

# Update. Update the object in place, then commit.
appt.title = 'Your Appointment'
session.commit()

# Delete. Tell the session to delete the object.
session.delete(appt)
session.commit()
```

Here are some sample queries you can run to get an idea of how SQLAlchemy works. Each `appt` example is a Python object of type `Appointment`. Each `appts` example is a Python list of `Appointment` objects.

```
# Get an appointment by ID.
appt = session.query(Appointment).get(1)

# Get all appointments.
appts = session.query(Appointment).all()

# Get all appointments before right now, after right now.
appts = session.query(Appointment).filter(Appointment.start <
datetime.now()).all()
appts = session.query(Appointment).filter(Appointment.start >=
datetime.now()).all()

# Get all appointments before a certain date.
appts = session.query(Appointment).filter(Appointment.start <=
datetime(2013, 5, 1)).all()

# Get the first appointment matching the filter query.
appt = session.query(Appointment).filter(Appointment.start <=
datetime(2013, 5, 1)).first()
```

SQLAlchemy provides full SQL functionality. Be sure to refer to the SQLAlchemy docs when you are generating SQL queries from your Python code.



About the `session` object: you will be using `db.session` (on the `db` object we added to `app.py` in this section) instead of `session` directly. Flask-SQLAlchemy will make sure that `db.session` is instantiated correctly and that data accessed through the database `session` in one web request does not interfere with the `session` in other requests.

Production database

SQLite is great for development and can work in production, but when you deploy your code, you may want to use PostgreSQL or MySQL as your database management system. You can change the underlying database while keeping your models as they are by specifying a different database URL and installing the necessary package to bind to the database.

For PostgreSQL:

- ▶ URL pattern: `postgresql://user:pass@localhost:5432/database_name`
- ▶ `pip install psycopg2`

For MySQL:

- ▶ URL pattern: `mysql://user:pass@localhost:3306/database_name`
- ▶ `pip install mysql-python`

Remember to create tables when you connect to a clean database.

SQLAlchemy knows how to translate data definitions to all major relational database management systems, and when you need to use specific database features, it has support for SQL dialects as documented at docs.sqlalchemy.org/en/rel_0_8/dialects/.

Naming conventions

Our `Appointment` class uses an underlying table named `appointment`. I recommend singular table names with a primary key, always named `id`, but you can decide for yourself how to name your tables and columns.

Custom SQL

At its core, SQLAlchemy is a Python toolkit to construct parameterized SQL expressions. At any point, if you need to write custom queries, you do not need to drop down to raw execute statement by passing strings—SQLAlchemy has the tools for you.

See the SQL Expression Language docs at docs.sqlalchemy.org/en/rel_0_8/core/.

Document stores and non-relational databases

If you prefer another database management system, perhaps one which is document-oriented instead of relational, use the tools available for connecting to that database from Python. As you can see, we have kept the domain model declarations isolated in one file and hooked them into the Flask application object with a few lines of code. Our application code throughout this book will use SQLAlchemy APIs to query data, but you can translate these calls to the database of your choice.

Handling forms and file uploads (Simple)

Let's get data from the user. Before we begin, remember to NEVER trust user input because sooner or later someone (or someone's script) with malicious intent will try to break your application.

Getting ready

We are working from the `forms.py` file inside the `sched` directory of our project.

How to do it...

1. Now that we have a data model, we need to present a form to the user in order to fill our database and validate user input to make sure it matches our schema. You can validate incoming data using any tools you like. We will use **WTForms** in our scheduler.
2. Update `requirements.txt`:

```
Flask
Flask-Script
Flask-SQLAlchemy
WTForms
```
3. Then:
\$ pip install -r requirements.txt
4. WTForms models forms with classes in a similar style to SQLAlchemy's declarative extension for modeling database tables. It takes the philosophy that forms should be modeled separately from the data, such that forms and data are separate concerns and often user forms (in the HTML user interface) do not line up exactly with the domain data (in the database modeled with SQLAlchemy). WTForms provides HTML form generation and validation of form data. In Flask, you will find submitted form data in `request.form` on the POST and PUT requests.

5. Here is our appointment's form. Each attribute takes a label and a set of validators. The `Length(max=255)` validator ensures an input of maximum of 255 characters, and the `required()` validator rejects an empty input. In `forms.py`:

```
from wtforms import Form, BooleanField, DateTimeField
from wtforms import TextAreaField, TextField
from wtforms.validators import Length, required

class AppointmentForm(Form):
    title = TextField('Title', [Length(max=255)])
    start = DateTimeField('Start', [required()])
    end = DateTimeField('End')
    allday = BooleanField('All Day')
    location = TextField('Location', [Length(max=255)])
    description = TextAreaField('Description')
```

How it works...

The purpose of `AppointmentForm` class is two-fold: render an input form in HTML and validate submitted data. This matches the SQLAlchemy-based `Appointment` class very closely. Where the `Appointment` model represents the domain and its persistence, this form class represents how to display a form in HTML and accept or reject the results.



Here is an opportunity to see what our code does before adding it to our Flask application, in our bottom-up approach. This section explains how WTForms works through some example code. You can add this code to the bottom of `forms.py` inside an `if name == '__main__':` block, and watch it run with `python forms.py`.

Before we add the form to our application in a later section, let's see the form in action on its own. We will print the values that WTForms creates so that we can understand what it is doing. If you are using Python 2, you will need to add a line to the top of your Python file in order to make your print expressions compatible between Python 2 and Python 3:

```
from __future__ import print_function
```

The following code prints the HTML representation of the `title` field:

```
# Demonstration of a WTForms form by itself.
form = AppointmentForm()
print('Here is how a form field displays:')
print(form.title.label)
print(form.title)
```

Output:

Here is how a form field displays:

```
<label for="title">Title</label>
<input id="title" name="title" type="text" value="">
```

When displayed in the browser, this form field renders as a very plain input as shown in the following screenshot. We will style our forms in a later section on HTML templating.



The following code shows how to interact with the form's validation directly. You can instantiate the `AppointmentForm` with a dictionary of form data. You can give Flask's `request.form` object to `AppointmentForm`, but here we will build our own dictionary to see how it works. Since HTML forms support multiple values for a single argument, we cannot use Python's built-in `dict`. Instead we use Werkzeug's `ImmutableMultiDict` type and make dummy data for a `title` field and omit all of the other fields.

```
# Give it some data.
from werkzeug.datastructures import \
    ImmutableMultiDict as multidict

data = multidict([('title', 'Hello, form!')])
form = AppointmentForm(data)
print('Here is validation...')
print('Does it validate: {}'.format(form.validate()))
print('There is an error attached to the field...')
print('form.start.errors: {}'.format(form.start.errors))
```

Output:

```
Here is validation...
Does it validate: False
There is an error attached to the field...
form.start.errors: [u'This field is required.']
```

Notice that when `AppointmentForm` is given the dictionary form data, `form.validate()` will process it and return either `True` or `False` depending on whether the form data is valid (`True` means valid). Errors from validators are loaded on each of the fields in an `errors` list so that you can display errors in context, which we will do when rendering a template in a later section.

There's more...

We handle file uploads a bit differently.

Handling file uploads

If you are handling file uploads, you need to interact with Flask directly. These snippets will get you started. First, the HTML form will include these basic elements: `enctype=multipart/form-data` attribute on the `form` element and file input in the form of "`<input type=file name=...>`".

```
<form method=post enctype=multipart/form-data>
  <input type=file name=file>
  <input type=submit value=Attach>
</form>
```

The code on the receiving end should look for a Werkzeug `FileStorage` object in the `request.files` dictionary by the name given on the file input element, and then sanitize the filename before saving the file to disk. Note that if you serve the file back with a `Content-Type` in the response which matches the filename's extension, you should only do so with file extensions you trust. Otherwise, an attacker could use your file-upload feature to embed JavaScript in your application, which will cause the user's browser to trust someone else's code.

```
import os

from flask import request
from werkzeug import secure_filename

# '...' here and below need to be set according to your project.
# In your configuration, set a root folder for file uploads.
FILE_FOLDER = '...'

@app.route('...', methods=['POST'])
def file_attach():
    # Match name from <input type=file name=file>: 'file'.
    filestorage = request.files['file']
    # Do not allow '..' in the filename.
    filename = secure_filename(filestorage.filename)
    dest = os.path.join(FILE_FOLDER, filename)
    filestorage.save(dest)
    return 'Put your response here.'
```

Templating with Jinja – setting a base template (Simple)

Flask ships with the Jinja templating engine to render any text format you need. Our scheduler will present HTML5 pages rendered by Jinja with a little style and a bit of JavaScript.

Getting ready

We will build templates using HTML, CSS, and JavaScript. You will need some familiarity with these to understand the responses we build in Flask. We will use Twitter's Bootstrap (getbootstrap.com/2.3.2/) framework for CSS and the jQuery (jquery.com) library for JavaScript. Both include online documentation. Bootstrap also includes icons, which we can use in our application, provided by Glyphicons (glyphicons.com). We will use a free theme for Bootstrap from Bootswatch (bootswatch.com).

We are working from `base.html` inside the `templates` directory within the `sched` directory.



This text shows templating by example. In our bottom-up approach, we will start with a base page structure before building out our specific application pages. Jinja has documentation on all of its template features. Be sure to refer to the docs at <http://jinja.pocoo.org/docs/> to check out what Jinja provides.

How to do it...

We will use public **content delivery networks (CDNs)** for Bootstrap and jQuery, which will speed up our start time in our project. NetDNA provides a public CDN for Bootstrap. The jQuery project provides a CDN through MediaTemple. Whenever you are ready or prefer to move away from a CDN, simply download the files you need and serve them with your application's static files, updating the links in the base template. The CDN URLs are, for use in the base template (where `... thing ...` identifiers are):

- ▶ `http://netdna.bootstrapcdn.com/bootswatch/2.3.2/united/bootstrap.min.css`
- ▶ `http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-responsive.min.css`
- ▶ `http://code.jquery.com/jquery-1.9.1.min.js`
- ▶ `http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/js/bootstrap.min.js`

Jinja supports template inheritance, where template files can extend an existing template. We will use this feature to layout our base structure, using the `block` template tag to indicate sections which child templates will fill. We provide a block to set the page title, add any additional content to the page head (which is useful for additional style and scripting), and fill the main content of the page. Jinja uses template tags `{% ... %}` to indicate Jinja-specific markup and directives. Here is our base template, in `templates/base.html`, with screenshots in the coming sections:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>{% block title %}{% endblock title %}</title>
  <link rel="shortcut icon" type="image/x-icon"
    href="{ { url_for('static', filename='img/favicon.ico') } }"/>
  <link href="... bootstrap css ..." rel="stylesheet">
  <link href="... responsive css ..." rel="stylesheet">
  <script src="... jquery js ..."></script>
  <script src="... bootstrap js ..."></script>
  {# Link any other .css or .js found in app static folder. #}
  <style>
    {# Add simple CSS style tweaks here. #}
  </style>
  <script>
    $(function() {
      {# Add page-load JavaScript here. #}
    });
  </script>
  {% block extra_head %}{% endblock extra_head %}
</head>
<body>
  <div id="main">
    <div class="utility-nav navbar navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          {# Navbar goes here. #}
        </div>
      </div>
    </div>
    <div class="content container">
      {% block main %}{% endblock main %}
    </div>
  </div>
</body>
</html>
```

How it works...

This base template allows us to write focused pages which share a common structure and style. Any template you create in the templates directory can inherit the base:

```
{% extends 'base.html' %}
{% block title %}Page Title{% endblock title %}
{% block main %}
    <h2>This is a child template.</h2>
{% endblock main %}
```

Render it with the `flask.render_template` function. If the file is named `index.html` inside the templates directory, you can render that template into a string with:

```
from flask import render_template
#...
@app.route('/')
def index():
    return render_template('index.html')
```

There's more...

There's a lot more that can be done using Jinja templating

Using a template engine other than Jinja

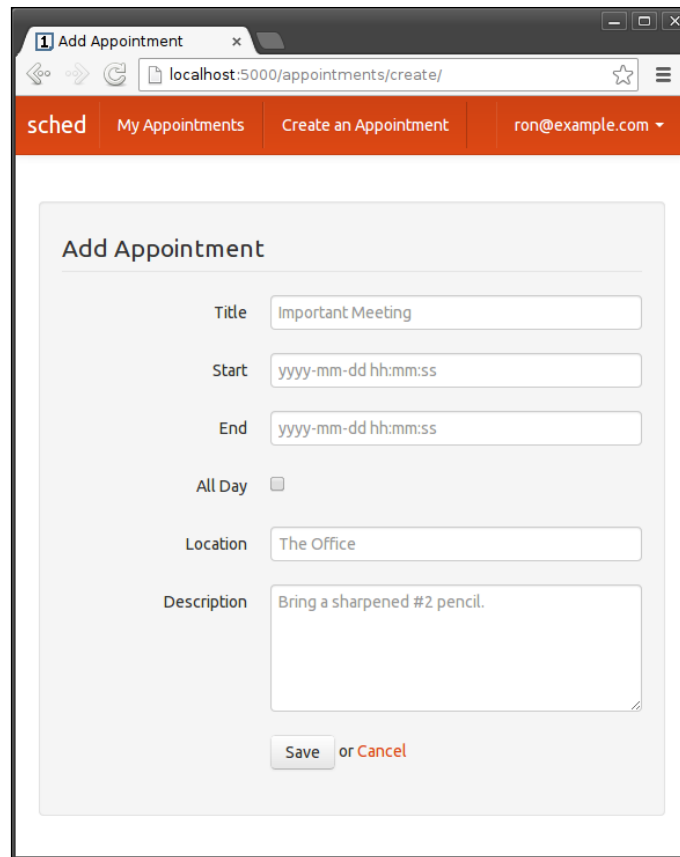
You can choose any template engine you like. When you call `render_template`, you have a Python string, and Flask converts the string into a `Response` object. If you prefer a different approach to templating, build a string and return it from your view functions. Flask bundles Jinja to provide a default templating environment for quick start, and for the community to build Flask extensions which are guaranteed a common environment.

Manage web packages with Bower

You can manage downloaded static files using Bower, a package manager for web tools, `bower.io`. With Bower, you would list dependencies using a `bower.json` file, use `bower install` to load these files, and then serve the files in your application's static area.

Creating a new record (Intermediate)

Here we provide a web form to create a new appointment as shown in the following screenshot:



The screenshot shows a web browser window with the title 'Add Appointment' and the address bar displaying 'localhost:5000/appointments/create/'. The browser's navigation bar includes links for 'sched', 'My Appointments', 'Create an Appointment', and a user profile 'ron@example.com'. The main content area is titled 'Add Appointment' and contains a form with the following fields:

- Title:** A text input field containing 'Important Meeting'.
- Start:** A date and time input field containing 'yyyy-mm-dd hh:mm:ss'.
- End:** A date and time input field containing 'yyyy-mm-dd hh:mm:ss'.
- All Day:** A checkbox that is currently unchecked.
- Location:** A text input field containing 'The Office'.
- Description:** A text area containing 'Bring a sharpened #2 pencil.'

At the bottom of the form, there are two buttons: 'Save' and 'Cancel'.

Getting ready

We will continue to work from the `app.py` file from `sched` and the `templates` directory.

How to do it...

1. We provide a view function to do both GET and POST handling for the form. Here, we pull together `db.session` from the database section and `AppointmentForm` from the forms section. In `app.py`:

```
from flask import abort, jsonify, redirect, render_template
from flask import request, url_for
from sched.forms import AppointmentForm
from sched.models import Appointment

# ... skipping ahead. Keep previous code from app.py here.

@app.route('/appointments/create/', methods=['GET', 'POST'])
def appointment_create():
    """Provide HTML form to create a new appointment."""
    form = AppointmentForm(request.form)
    if request.method == 'POST' and form.validate():
        appt = Appointment()
        form.populate_obj(appt)
        db.session.add(appt)
        db.session.commit()
        # Success. Send user back to full appointment list.
        return redirect(url_for('appointment_list'))
    # Either first load or validation error at this point.
    return render_template('appointment/edit.html',
        form=form)
```

2. Before we build out the input fields, we can create a utility for ourselves which will display all of the WTForms features for a given field: label, input, and errors. Jinja has macros, which are similar to Python functions. We will create a macro to render an edit field from `AppointmentForm`.
3. For inputs, we can follow Bootstrap conventions with `control-group` and `controls` page elements, which will let us completely control the form flow from CSS. We want to start a new template with our macro, so that we can reuse it in other templates we create. In `templates/appointment/common.html`:

```
{% macro edit_field(field, catch_kwargs=true) %}
<div class="control-group{% if field.errors %} error{% endif
%}">
  {{ field.label(class="control-label") }}
<div class="controls">
  {{ field(**kwargs) }}
  {% for error in field.errors %}
    <span class="help-inline">{{ error }}</span>
  {% endfor %}
</div>
</div>
{% endmacro %}
```

Now we can build a form using our new macro. Starting a new template, we can extend the base and import the macro. When extending the base, we provide blocks title and main, using the same syntax as the base template. You can set variables within a template, for use within that template, using `{% set ... %}`.

4. In `templates/appointment/edit.html`:

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

{% from 'appointment/common.html' import edit_field %}

{% set title = 'Add Appointment' %}

{% block title %}{{ title }}{% endblock title %}

{% block main %}
<div class="row">
  <div class="appointment-edit well offset2 span8">
    <form method="post" class="form-horizontal">
      <legend>{{ title }}</legend>
      {{ edit_field(form.title, maxlength=255, class="span3",
        placeholder="Important Meeting") }}
      {{ edit_field(form.start, class="span3",
        type="datetime",
        placeholder="yyyy-mm-dd hh:mm:ss") }}
      {{ edit_field(form.end, class="span3", type="datetime",
        placeholder="yyyy-mm-dd hh:mm:ss") }}
      {{ edit_field(form.allday) }}
      {{ edit_field(form.location, maxlength=255,
        class="span3",
        placeholder="The Office") }}
      {{ edit_field(form.description, rows="5",
        class="span5",
        placeholder="Bring a sharpened #2 pencil.") }}
      <div class="form-actions">
        <button type="submit" class="btn">Save</button>
        or <a href="{{ url_for('appointment_list') }}">Cancel</a>
      </div>
    </form>
  </div>
</div>
{% endblock main %}
```

How it works...

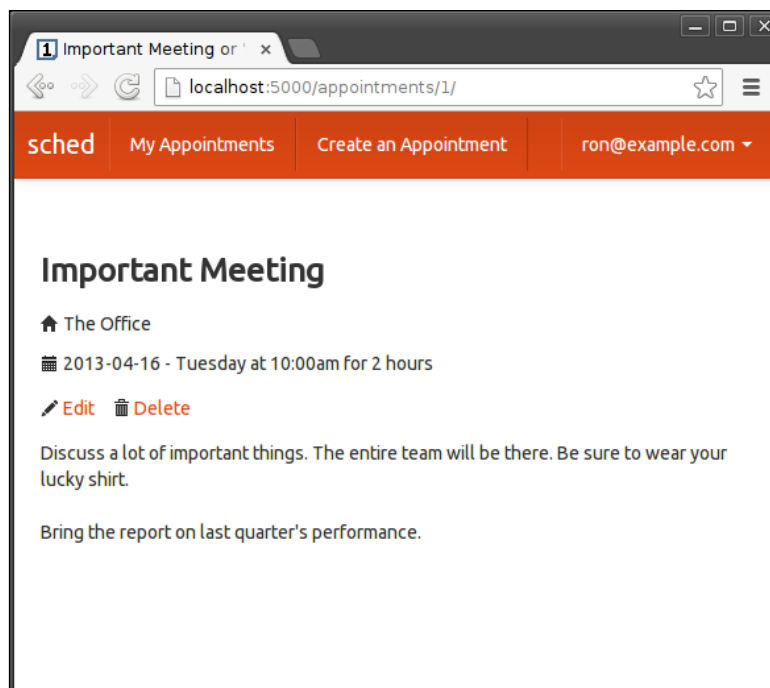
On GET, the form is rendered with an action to POST the data to the same URL. On POST, the view function validates the data and adds it to the database. If the validation fails, the POST renders the template again, but this time the form object has errors.

Jinja uses template syntax `{{ ... }}` to print a Python object in context to the template's output. We use a simple `if` statement to add the error class if the field has errors, which will highlight the input when errors exist (thanks to Bootstrap style). We use a `for` loop to lay down a help span for each error on the field. The use of `**kwargs` will catch all arguments which are given beyond the macro's call signature; this lets us pass in all WTForm field options through the macro. The `kwargs` feature only works in Jinja macros when `catch_kwargs=True`.

CSS classes `form-horizontal`, `span3`, and `span5` tell Bootstrap how to layout the form on its grid system. The `placeholder` attribute is a feature in HTML5 to show a watermark in the input when there is no content.

Displaying a record (Intermediate)

With content going into the database, let's get it back out as shown in the following figure:



Getting ready

We will continue to work from the `app.py` file from the `sched` folder and the `templates` directory.

How to do it...

1. We provide a simple query to get an appointment by database ID. In `app.py`:

```
# Note the import statements from the previous section.

@app.route('/appointments/<int:appointment_id>/')
def appointment_detail(appointment_id):
    """Provide HTML page with a given appointment."""
    # Query: get Appointment object by ID.
    appt = db.session.query(Appointment).get(appointment_id)
    if appt is None:
        # Abort with Not Found.
        abort(404)
    return render_template('appointment/detail.html',
                           appt=appt)
```

2. Let's take the same macro approach to displaying an appointment. A template macro will give us a tool to display an appointment any time we need it. In `templates/appointment/common.html`:

```
{% macro detail(appt,
    link_title=false,
    show_edit=true,
    show_description=true) %}
<div class="appointment-detail">
    {% if link_title %}
        <h3>
            <a href="{{ url_for('appointment_detail',
                appointment_id=appt.id) }}">
                {{ appt.title }}</a>
        </h3>
    {% else %}
        <h3>{{ appt.title }}</h3>
    {% endif %}
    {% if appt.location %}
        <p><i class="icon-home"></i> {{ appt.location }}</p>
    {% endif %}
```

```
{% if appt.allday %}
    <p><i class="icon-calendar"></i> {{ appt.start | date
    }}</p>
{% else %}
    <p><i class="icon-calendar"></i> {{ appt.start |
    datetime }}
    for {{ appt.duration | duration }}</p>
{% endif %}
{% if show_edit %}
<div class="edit-controls">
    <i class="icon-pencil"></i>
    <a href="{{ url_for('appointment_edit',
    appointment_id=appt.id) }}">Edit</a>
    <span class="inline-pad"></span>
    <i class="icon-trash"></i>
    <a class="appointment-delete-link" href="#"
    data-delete-url="{{ url_for('appointment_delete',
    appointment_id=appt.id) }}">Delete</a>
</div>
{% endif %}
{% if show_description and appt.description %}
<div class="row">
    <p class="span5">{{ appt.description | nl2br }}</p>
</div>
{% endif %}
</div>
{% endmacro %}
```

3. We can then use the macro in templates/appointment/detail.html:

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

{% from 'appointment/common.html' import detail %}

{% block title %}
    {{ appt.title }}
{% endblock title %}

{% block main %}
<div class="row">
    <div class="span12">
        {{ detail(appt) }}
    </div>
</div>
{% endblock main %}
```

How it works...

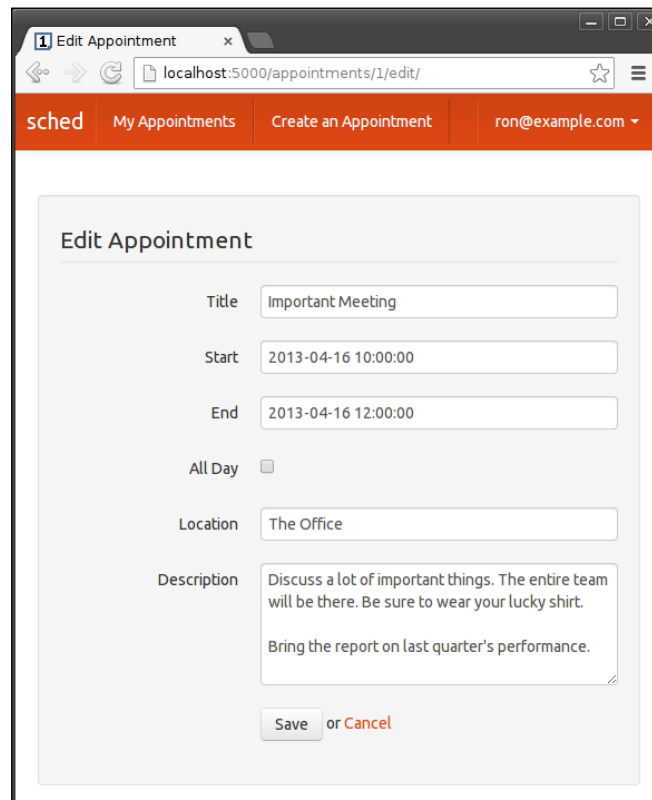
If a request comes in for a database ID that does not exist, we tell Flask to abort with a **404 Not Found** response. The `flask.abort` function is implemented as an exception, so Python will stop execution in the current function when `abort` is called.

We will also implement a few utilities for Jinja named filters, which formats the output of a Python object before it goes into the template's output, in the form of `{{ foo | filter }}`. We will build filters `date`, `datetime`, `duration`, and `nl2br`. On your first pass through in implementing these, simply omit these from your template; use `{{ appt.start }}` instead of `{{ appt.start | date }}`. This will give you a clearer idea of why we are building the filter.

For individual fields which may not exist, we use Python's `or` behavior. When used for display or assignment, `or` expression is "shortcut" until a true value is hit, and that value is used. This approach lets us provide a default display value in a simple way.

Editing a record (Intermediate)

Now we provide an edit page for existing appointments, as shown in the following screenshot:



The screenshot shows a web browser window with the title 'Edit Appointment' and the URL 'localhost:5000/appointments/1/edit/'. The browser's address bar shows the URL. The page has a navigation bar with 'sched' (active), 'My Appointments', 'Create an Appointment', and 'ron@example.com'. The main content area is titled 'Edit Appointment' and contains a form with the following fields:

- Title: Important Meeting
- Start: 2013-04-16 10:00:00
- End: 2013-04-16 12:00:00
- All Day: ☐
- Location: The Office
- Description: Discuss a lot of important things. The entire team will be there. Be sure to wear your lucky shirt. Bring the report on last quarter's performance.

At the bottom of the form are 'Save' and 'Cancel' buttons.

Getting ready

We will continue to work from the `app.py` file from the `sched` directory and `templates` directory.

How to do it...

1. The edit workflow is a mix of add and detail views. We get an appointment, and if it exists, we display a form to edit it. In `app.py`:

```
@app.route('/appointments/  
    <int:appointment_id>/edit/',  
    methods=['GET', 'POST'])  
  
def appointment_edit(appointment_id):  
    """Provide HTML form to edit a given appointment."""  
    appt = db.session.query(Appointment).get(appointment_id)  
    if appt is None:  
        abort(404)  
    form = AppointmentForm(request.form, appt)  
    if request.method == 'POST' and form.validate():  
        form.populate_obj(appt)  
        db.session.commit()  
        # Success. Send the user back to the detail view.  
        return redirect(url_for('appointment_detail',  
                                appointment_id=appt.id))  
    return render_template('appointment/edit.html',  
                           form=form)
```

2. In `templates/appointment/edit.html`:

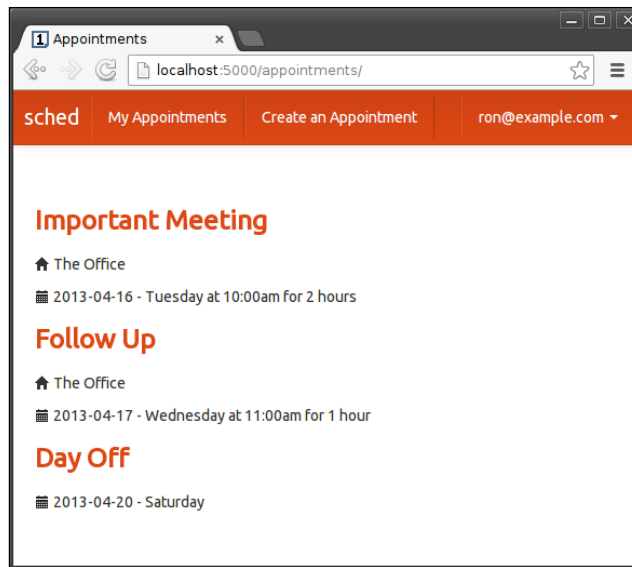
```
{% if request.endpoint.endswith('edit') %}  
    {% set title = 'Edit Appointment' %}  
{% else %}  
    {% set title = 'Add Appointment' %}  
{% endif %}
```

How it works...

We reuse the same template as the create form, but we say **Edit** instead of **Add**. Since this is very simple logic, we can use an `if` statement in the template. Flask lets us inspect the request object in the template. If you find your project does a lot of request inspection, you should consider a design pattern which moves this logic back into your Python code.

Listing all records (Simple)

We don't want users to keep track of the database IDs of their appointments, so we provide a view, which will list all the appointments that they have, as shown in the following screenshot:



Getting ready

We will continue to work from the `app.py` file from the `sched` directory and the `templates` directory.

How to do it...

1. We query for all appointments, in `app.py`:

```
@app.route('/appointments/')
def appointment_list():
    """Provide HTML listing of all appointments."""
    # Query: Get all Appointment objects, sorted by date.
    appts = (db.session.query(Appointment)
              .order_by(Appointment.start.asc()).all())
    return render_template('appointment/index.html',
                           appts=appts)
```


2. In `templates/appointment/index.html`:

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

{% from 'appointment/common.html' import detail %}

{% block title %}Appointments{% endblock title %}

{% block main %}
<div class="row">
  {% for appt in appts %}
    <div class="span12">
      {{ detail(appt,
        link_title=true,
        show_edit=false,
        show_description=false) }}
    </div>
  {% else %}
    <h3 class="span12">No appointments.</h3>
  {% endfor %}
</div>
{% endblock main %}
```

How it works...

We query for all appointments in ascending order by appointment start time. SQLAlchemy declarative classes include helpers on column attributes that let us provide SQL clauses, here with `.asc()` to indicate the `sort` field. In the appointment list template, we iterate over all appointment records and reuse our display macro and switch off the features which only apply to the detail view, using the call signature we created.

There's more...

We kept the appointment list simple. If you are new to database identifiers, you may be wondering how the appointment URLs are generated.

Database identifiers

A note on those database IDs: they are system generated by our database management system and SQLAlchemy. That is, they are meaningless outside of our application. IDs make for good URLs, but not for human memory. This view is one where you can get creative and use the day, week, month, and year calendar metaphors. You can do that, but that will live mostly in HTML, CSS, and JavaScript. For our simple scheduler application, we just list out the appointments in a list.

Deleting a record (Advanced)

We are all set with our appointment interactions. Wait; how do we delete appointments?

Getting ready

We will continue to work from `sched's app.py` file and `templates` directory.

How to do it...

1. We build an Ajax handler, in `app.py`:

```
@app.route('/appointments/
<int:appointment_id>/delete/',
methods=['DELETE'])

def appointment_delete(appointment_id):
    """Delete record using HTTP DELETE, respond with JSON."""
    appt = db.session.query(Appointment).get(appointment_id)
    if appt is None:
        # Abort with Not Found, but with simple JSON response.
        response = jsonify({'status': 'Not Found'})
        response.status = 404
        return response
    db.session.delete(appt)
    db.session.commit()
    return jsonify({'status': 'OK'})
```

2. A little jQuery will wire up all delete links to use this handler, in `templates/base.html`:

```
<script>
$(function() {
  $(".appointment-delete-link").on("click", function() {
    var delete_url = $(this).attr('data-delete-url');
    $.ajax({
      url: delete_url,
      type: 'DELETE',
      success: function(response) {
        if (response.status == 'OK') {
          window.location = {{ url_for('appointment_list') }};
        } else {
          alert('Delete failed.')
        }
      }
    })
  })
})
```

```
    }  
    });  
    return false;  
  });  
});  
</script>
```

How it works...

We provide our first view function which does not render HTML, but uses a minimal Ajax interacting with JSON. It accepts HTTP DELETE requests and does the work. We do not have to expose delete functionality through GET requests, which would let users accidentally delete database records by browsing. This is particularly important when we publish code which can be hit by search engines and other robots. Crawling all of our pages would delete our entire database! Flask provides a `jsonify` function to turn a Python dictionary into a JSON response.

How do you get a browser to send a DELETE request? With JavaScript. The jQuery library makes Ajax calls a lot simpler than JavaScript alone. We add a jQuery hook which will take all delete links and submit an ajax call when they are clicked. The on-click callback grabs the deletion URL from the delete link, and sends it as a DELETE request. On success, it redirects the current browser window to the appointment list. By placing this script into the base template, it will make all delete links functional on the appointment list and detail pages.

There's more...

We only cover Ajax briefly here; it deserves an entire book.

Ajax

Ajax simply means that we render a web page and have the browser communicate with our web service without necessarily having to reload the page. That is, render the page once, and update the content displayed on the page without refreshing the entire page. This was originally called AJAX for "asynchronous JavaScript and XML", but has become a common word among web developers and can mean any serialization format, here JSON instead of XML.

Using custom template filters in Jinja (Advanced)

To wrap up our templates, we need to clean up the display of some of our fields.

Getting ready

We move to the `filters.py` file from the `sched` directory and work with `app.py` again.

How to do it...

1. We want to display time with a clean format. In `filters.py`:

```
def do_datetime(dt, format=None):
    """Jinja template filter to format a datetime object."""
    if dt is None:
        # By default, render an empty string.
        return ''
    if format is None:
        # No format is given in the template call.
        # Use a default format.
        #
        # Format time in its own strftime call in order to:
        # 1. Left-strip leading 0 in hour display.
        # 2. Use 'am'/'pm' (lower case) instead of 'AM'/'PM'.
        formatted_date = dt.strftime('%Y-%m-%d - %A')
        formatted_time = \
            dt.strftime('%I:%M%p').lstrip('0').lower()
        formatted = '%s at %s' % \
            (formatted_date, formatted_time)
    else:
        formatted = dt.strftime(format)
    return formatted
```

2. Add more filters here, and provide a hook to initialize the Flask application.

```
def init_app(app):
    """Initialize a Flask application with custom filters."""
    app.jinja_env.filters['datetime'] = do_datetime
```

3. Use the initialization hook in `app.py`:

```
from sched import filters
filters.init_app(app)
```

How it works...

Jinja supports an environment where Python functions are provided in the template as filters. We have a `datetime` field on our appointment, so we provide some formatting for it. You can call attributes and methods directly within the template, `{{ dt.strftime('%Y-%m-%d') }}`, but defining a filter lets us specify how to format all dates centrally while still exposing parameters. You can call `{{ dt | datetime }}` with our `datetime` filter to get the default functionality that we defined, as well as `{{ dt | datetime('%Y-%m-%d') }}` where the argument in the template call is passed in as an argument after the value being filtered.

There's more...

You can find the full code for filters in the source code files downloadable on the Packt website.

Additional filters

The full source code of `sched` includes filters for default date formatting, duration formatting for the appointment records, and the `nl2br` filter to preserve user-entered line breaks in the appointment description field (since HTML normalizes whitespace otherwise).

Sending error responses (Simple)

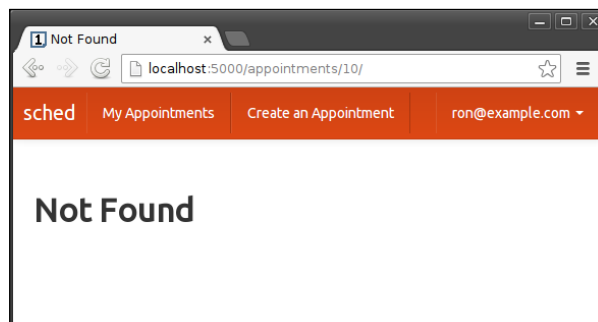
In this section, we will send an error page to the user with style.

Getting ready

We are working from the `app.py` file inside the `sched` directory of our project, and in the `templates` directory inside `sched`. Create a directory named `error` inside the `templates` directory.

How to do it...

1. When the user hits a **Not Found** page, we want to display a page that will direct them back to the appointment list. You can tell Flask how to render responses for error cases, typically for HTTP error codes. We will provide a custom 404 Not Found page as shown in the following screenshot:



2. Flask provides `app.errorhandler` to tell Flask how to handle certain kinds of errors, taking either 4xx or 5xx HTTP status codes or Python exception classes. Decorate a function which accepts an `Exception` instance and returns a response. Be sure to include the HTTP status code in the Flask response, by providing 404 in a tuple response with the rendered template string. In the following code, we render a not-found template, which provides a link back to the appointment list. In `app.py`:

```
@app.errorhandler(404)
def error_not_found(error):
    return render_template('error/not_found.html'), 404
```

3. In `templates/error/not_found.html`:

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

{% block title %}Not Found{% endblock title %}

{% block main %}
<h2>Not Found</h2>
{% endblock main %}
```

The navbar element in the base template will help users navigate back to a known page.

How it works...

The `app.errorhandler` decorator accepts exception classes or HTTP status codes as integers. Following are some HTTP status codes to get you started. You can use these with `flask.abort(status_code)` in your view functions to jump directly to an error response, and define a custom `app.errorhandler` to provide a styled response.

- ▶ 400 Bad Request
- ▶ 401 Unauthorized
- ▶ 403 Forbidden
- ▶ 404 Not Found
- ▶ 405 Method Not Allowed (you may have forgotten methods in your route)
- ▶ 410 Gone (and not coming back)
- ▶ 500 Internal Server Error

See www.w3.org/Protocols/rfc2616/rfc2616-sec10.html for the full list.

The 500 error handler will display anytime your code has an uncaught exception.

There's more...

Flask provides a very useful debugger to inspect your errors in development.

Handling specific exceptions

You can provide a custom error page on Python exceptions:

```
class SchedulingException(Exception):
    """Some application-specific error has occurred."""
    pass

@app.errorhandler(SchedulingException)
def scheduling_exception_handler(error):
    return 'Just a demo of app.errorhandler...', 500

@app.route('/test_error/')
def test_error():
    raise SchedulingException('Use my custom error handler.')
```

Flask's debugger

This is a good place to discuss Flask's debugger. When a Flask application has `app.config['DEBUG']` set to `True`, any uncaught exception in the application code will display a developer-friendly debugger in the browser. This debugger makes the Python stack trace interactive, letting you run code at any point in the stack frame and viewing code in context.

In `app.py`:

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

You can trigger exceptions intentionally for a quick peek at your stack. Write in an exception and hit the route in your web browser. My two favorite exceptions are:

```
1/0 # Divide by zero to get an exception.
raise Exception("Don't Panic!")
```

Only use this in development on a secured machine, since the interactive interpreter allows execution of Python code from the browser. If the interactive interpreter becomes unresponsive in the browser, either the development server has shut down or has moved on to new requests, forgetting about your error.

Authenticating users (Advanced)

So far we have assumed that anyone who can access the `sched` application should be able to create, display, edit, list, and delete appointment records in the database. Our Flask application needs to authenticate users and protect our database.

Getting ready

We return to the `models.py` file inside the `sched` directory of our project.

How to do it...

1. We start with a database record for the user with a unique e-mail address, in `models.py`:

```
from datetime import datetime

from sqlalchemy import Column
from sqlalchemy import Boolean, DateTime, Integer, String
from sqlalchemy.orm import synonym
from sqlalchemy.ext.declarative import declarative_base
from werkzeug import check_password_hash
from werkzeug import generate_password_hash

Base = declarative_base()

class User(Base):
    """A user login, with credentials and authentication."""
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    created = Column(DateTime, default=datetime.now)
    modified = Column(DateTime, default=datetime.now,
                       onupdate=datetime.now)

    name = Column('name', String(200))
    email = Column(String(100), unique=True, nullable=False)
    active = Column(Boolean, default=True)
```


2. Then, we add a means to store a password hash (continuing in User definition):

```
_password = Column('password', String(100))

def _get_password(self):
    return self._password

def _set_password(self, password):
    if password:
        password = password.strip()
        self._password = generate_password_hash(password)

password_descriptor = property(_get_password,
                                _set_password)
password = synonym('_password',
                    descriptor=password_descriptor)
```

3. Finally, we add a means to validate a password (continuing in User definition):

```
def check_password(self, password):
    if self.password is None:
        return False
    password = password.strip()
    if not password:
        return False
    return check_password_hash(self.password, password)

@classmethod
def authenticate(cls, query, email, password):
    email = email.strip().lower()
    user = query(cls).filter(cls.email==email).first()
    if user is None:
        return None, False
    if not user.active:
        return user, False
    return user, user.check_password(password)
```

How it works...

We need a place to store user records, so that we can verify whether an authentication request is for a valid user in our application. However, we do not want to store the password in clear text, which would let anyone with read access to the database table know how to login as a valid user in the application. This may not seem important the first time you come across the idea, but you do not want to handle the responsibility of storing everyone's passwords, especially if users reuse passwords across applications. (This should be a hint as to why that is a bad idea.)

Werkzeug provides utilities to hash a password and verify that the provided password matches that hash. We store the hash and not the clear-text password. Anytime `user.password` is set on a `User` instance, the password is immediately hashed using a Python descriptor and SQLAlchemy's synonym hook. The synonym lets us have `user.password` behave just like every other SQLAlchemy object attribute, but when `user.password` is accessed, getting the value will call `_get_password` and setting it will call `_set_password`, which will store a hashed value.

We can now authenticate using e-mail, password, and a SQLAlchemy query object with `User.authenticate(query, email, password)`, which returns a `user, bool` pair (tuple), where `user` is the matching user object and the Boolean value indicates whether the authentication is valid.

There's more...

We need to integrate our e-mail/password authentication into our application.

Appointment relationship

We can build a relationship between appointments and users so that we display only the current user's appointments when we load schedules. Update the `Appointment` class in `models.py` to include a foreign-key relationship to `User`:

```
class Appointment(Base):
    # ... existing code.
    user_id = Column(Integer,
        ForeignKey('user.id'), nullable=False)
    user = relationship(User, lazy='joined', join_depth=1)
    # ... existing code.
```

Accessing an `Appointment` object will now provide the `appointment.user` property to access the `User` object for the owner of that appointment. Because we used `ForeignKey`, SQLAlchemy knows how to fetch the related `User` record when accessing an `Appointment` object.

Opening a session

HTTP is stateless, meaning that one request has no information about the previous requests. We need to open a session that is retained from request-to-request, so that we can authenticate the user in one request and use that authentication in each of the following requests that the user makes in a given sitting.

In the next section, we will see how to handle the session, adding a `current_user` object to our code which will represent the `User` database object which matches the currently authenticated user. Now that we've added a relationship between `User` and `Appointment` in `models.py`, we need to update `app.py` for every case where we create or retrieve an `Appointment` object. Change the `Appointment` object creation line to:

```
appt = Appointment(user_id=current_user.id)
```

Then verify whether the current `Appointment` is for the current `User` when retrieving records:

```
if appt.user_id != current_user.id:
    abort(403)
```

Database migration

Changing `models.py` means that we need to update the database structure of the database we have been using in development, or the production database if we have already deployed the `sched` application. If the only changes we have made are to add tables, then we can simply make another call to `Base.metadata.create_all(engine)`.

Sometimes in development, this is the easiest method: just destroy the development database, run `create_all` again, then start adding data again. Setting up initial development or test data can be cumbersome, so you might want to invest in **fixtures**, which you load before running your application development server. There are several projects in the Python community to do this, including `fixture` at pypi.python.org/pypi/fixture.

For column changes, we need to provide an automation to change the live database, especially for production databases, which people are using. If you are familiar with SQL, you can write your own `ALTER TABLE` statements and provide your own scripts. If you would like a toolkit to manage this, SQLAlchemy's creator Mike Bayer has written Alembic as a tool to perform migrations in SQLAlchemy, alembic.readthedocs.org.

Authenticating with existing services

How do new users get started? With user authentication in place for `sched`, you need to provide a means to register users with your application. This can be as simple as creating `User` objects from the Python interactive interpreter, or it could be a user signup form within your application.

Alternatively, you can reduce the `User` implementation to the core application data that you need and use an existing service to authenticate your users externally, creating new `User` objects on the first such authentication. This will keep users from having to remember yet another password and speed up your user acquisition process.

OpenID (openid.net) provides an open standard to perform this style of authentication. Popular services such as Google have their own documentation for authentication and authorization using existing user accounts, developers.google.com/accounts/.

Handling sessions and users (Intermediate)

Since HTTP is stateless, we need to track some data across requests with a session.

Getting ready

We will continue to work from the `app.py` file from the `sched` directory and the `models.py` file.

How to do it...

1. Flask provides a `session` object, which behaves like a Python dictionary, and persists automatically across requests. You can, in your Flask application code:

```
from flask import session
# ... in a request ...
session['spam'] = 'eggs'
# ... in another request ...
spam = session.get('spam') # 'eggs'
```

2. Flask-Login provides a simple means to track a user in Flask's session. Update `requirements.txt`:

```
Flask
Flask-Login
Flask-Script
Flask-SQLAlchemy
WTForms
```

3. Then:

```
$ pip install -r requirements.txt
```

4. We can then load Flask-Login into sched's request handling, in `app.py`:

```
from flask.ext.login import LoginManager, current_user
from flask.ext.login import login_user, logout_user
from sched.models import User

# Use Flask-Login to track current user in Flask's session.
login_manager = LoginManager()
login_manager.setup_app(app)
login_manager.login_view = 'login'

@login_manager.user_loader
def load_user(user_id):
    """Flask-Login hook to load a User instance from ID."""
    return db.session.query(User).get(user_id)
```

5. Flask-Login requires four methods on the User object, inside class `User` in `models.py`:

```
def get_id(self):
    return str(self.id)

def is_active(self):
    return True

def is_anonymous(self):
    return False

def is_authenticated(self):
    return True
```



Flask-Login provides a **UserMixin** (`flask.ext.login.UserMixin`) if you prefer to use its default implementation.



6. We then provide routes to log the user in when authenticated and log out. In `app.py`:

```
@app.route('/login/', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated():
        return redirect(url_for('appointment_list'))
    form = LoginForm(request.form)
    error = None
    if request.method == 'POST' and form.validate():
```

```

        email = form.username.data.lower().strip()
        password = form.password.data.lower().strip()
        user, authenticated = \
            User.authenticate(db.session.query, email,
                             password)
        if authenticated:
            login_user(user)
            return redirect(url_for('appointment_list'))
        else:
            error = 'Incorrect username or password.'
            return render_template('user/login.html',
                                   form=form, error=error)

@app.route('/logout/')
def logout():
    logout_user()
    return redirect(url_for('login'))

```

7. We then decorate every view function that requires a valid user, in `app.py`:

```

from flask.ext.login import login_required
@app.route('/appointments/')
@login_required
def appointment_list():
    # ...

```

How it works...

On `login_user`, Flask-Login gets the user object's ID from `User.get_id` and stores it in Flask's session. Flask-Login then sets a `before_request` handler to load the user instance into the `current_user` object, using the `load_user` hook we provide. The `logout_user` function then removes the relevant bits from the session.

If no user is logged in, then `current_user` will provide an anonymous user object which results in `current_user.is_anonymous()` returning `True` and `current_user.is_authenticated()` returning `False`, which allows application and template code to base logic on whether the user is valid. (Flask-Login puts `current_user` into all template contexts.) You can use `User.is_active` to make user accounts invalid without actually deleting them, by returning `False` as appropriate.

View functions decorated with `login_required` will redirect the user to the login view if the current user is not authenticated, without calling the decorated function.

There's more...

Flask's session supports display of messages and protection against request forgery.

Flashing messages

When you want to display a simple message to indicate a successful operation or a failure quickly, you can use Flask's flash messaging, which loads the message into the session until it is retrieved. In application code, inside request handling code:

```
from flask import flash
flash('Sucessfully did that thing.', 'success')
```

In template code, where you can use the 'success' category for conditional display:

```
{% for cat, m in get_flashed_messages(with_categories=true) %}
  <div class="alert">{{ m }}</div>
{% endfor %}
```

Cross-site request forgery protection

Malicious web code will attempt to forge data-altering requests for other web services. To protect against forgery, you can load a randomized token into the session and into the HTML form, and reject the request when the two do not match. This is provided in the Flask-SeaSurf extension, pythonhosted.org/Flask-SeaSurf/ or the Flask-WTF extension (which integrates WTForms), pythonhosted.org/Flask-WTF/.

Deploying to the world (Advanced)

Once satisfied with your application, you can deploy your application which would be available to the world.

Getting ready

You need a computer which is online and accessible to your target users. Install Python and place `requirements.txt` in a folder where you have command-line access, just as we did for your development environment (virtualenv is production-appropriate), then create your database tables. You can deploy to any operating system: Windows and any Unix-like systems (including Mac OS X).

How to do it...

1. From a clean Ubuntu 12.04 server installation:

```
$ sudo apt-get update && sudo apt-get install nginx
$ # Load config into /etc/nginx/sites-available/default
$ sudo service nginx restart
$ pip install gunicorn
$ gunicorn --access-logfile - -b 127.0.0.1:5000 sched.app:app
```

2. Use the `pip` and `gunicorn` programs in `virtualenv`. Load into the `nginx` configuration, changing `localhost` to your domain name if you have one:

```
server {
    listen 80;
    server_name localhost;
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
    location / {
        proxy_pass http://127.0.0.1:5000/;
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

3. Then add the `gunicorn` process into the server's boot order instead of running it by hand. See your operating system's documentation. For Unix-like systems, `supervisord` provides management of long-running applications, supervisord.org.

How it works...

Anytime `nginx` receives a matching request, it sends a proxy request to the HTTP server running on port 5000, which is `gunicorn` in our application, and passes along the response. Note that `gunicorn` does not automatically reload your application on code updates. You can reload with process signals as described in docs.gunicorn.org/en/latest/faq.html.

See docs.gunicorn.org/en/latest/deploy.html for additional documentation.

There's more...

Deployment has many options; the configuration is entirely up to your project.

Handling static files with nginx

Flask provides a static file handler for convenience. Now that we have an optimized HTTP server in place, we can have it serve static files directly. The following configuration block shows how to serve static files with nginx and proxy all other requests to Flask. Adjust the configuration according to your setup:

- ▶ Change `/var/www/myproject` to the filepath containing the `sched` folder from our project layout.
- ▶ Change `/static/` if you changed the default static route in Flask.

The updated server configuration, to handle static files:

```
server {
    listen 80;
    server_name localhost;
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
    location /static/ {
        # /static/ is appended to root path on the request.
        root /var/www/myproject/sched/;
    }
    location / {
        proxy_pass http://127.0.0.1:5000/;
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Deployable on any OS with HTTP proxying

I prefer to run applications in production using HTTP proxying with application code served by stand-alone, independent operating system processes. Most industrial strength HTTP servers include a means to proxy requests to another web service, just as we demonstrated with nginx. gunicorn only runs on Unix-like systems, but cherrypy (cherrypy.org) provides a cross-platform WSGI server. You can start your application with a script which follows this example:

```
from cherrypy.wsgiserver import CherryPyWSGIServer
from sched.app import app
server = CherryPyWSGIServer(('127.0.0.1', 5000), app)
server.start()
```

Windows server deployment

To run your Python application directly within Windows Server IIS, see the NWSGI project at nwsgi.codeplex.com.

Other deployment options

See the Flask docs at flask.pocoo.org/docs/deploying/.



Thank you for buying Instant Flask Web Development

About Packt Publishing

Packt, pronounced 'packed', published its first book *"Mastering phpMyAdmin for Effective MySQL Management"* in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0

Paperback: 488 pages

Over 110 easy-to-follow recipes backed up with real-life examples, walking you through basic Ext JS features to advanced application design using Sencha's Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips



Responsive Web Design with HTML5 and CSS3

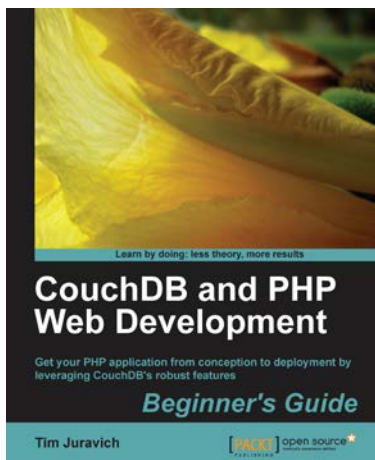
ISBN: 978-1-84969-318-9

Paperback: 324 pages

Learn responsive design using HTML5 and CSS3 to adapt websites to any browser or screen size

1. Everything needed to code websites in HTML5 and CSS3 that are responsive to every device or screen size
2. Learn the main new features of HTML5 and use CSS3's stunning new capabilities including animations, transitions and transformations
3. Real world examples show how to progressively enhance a responsive design while providing fall backs for older browsers

Please check www.PacktPub.com for information on our titles



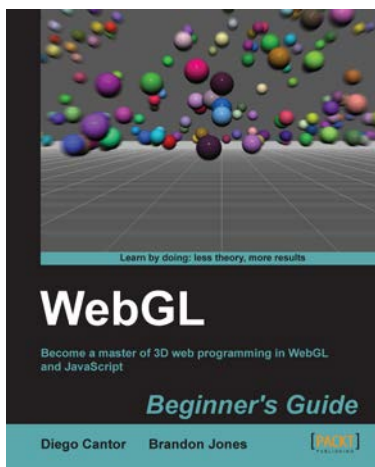
CouchDB and PHP Web Development Beginner's Guide

ISBN: 978-1-84951-358-6

Paperback: 304 pages

Get your PHP application from conception to deployment by leveraging CouchDB's robust features

1. Build and deploy a flexible Social Networking application using PHP and leveraging key features of CouchDB to do the heavy lifting
2. Explore the features and functionality of CouchDB, by taking a deep look into Documents, Views, Replication, and much more.
3. Conceptualize a lightweight PHP framework from scratch and write code that can easily port to other frameworks



WebGL Beginner's Guide

ISBN: 978-1-84969-172-7

Paperback: 376 pages

Become a master of 3D web programming in WebGL and JavaScript

1. Dive headfirst into 3D web application development using WebGL and JavaScript.
2. Each chapter is loaded with code examples and exercises that allow the reader to quickly learn the various concepts associated with 3D web development
3. The only software that the reader needs to run the examples is an HTML5 enabled modern web browser. No additional tools needed.

Please check www.PacktPub.com for information on our titles