



Part 2: Web Development with Python

Table of Contents

Introduction	1.1
Getting Started	1.2
Python Review	1.2.1
Development Environments	1.2.2
SQLite	1.2.3
pip	1.2.4
virtualenv	1.2.5
Web Browsers	1.2.6
Version Control	1.2.7
Interlude: Modern Web Development	1.3
Front-end, Back-end, and Middleware	1.3.1
Model-View-Controller	1.3.2
Flask: Quick Start	1.4
Overview	1.4.1
Installation	1.4.2
Hello World	1.4.3
App Flow	1.4.4
Dynamic Routes	1.4.5
Response Object	1.4.6
Debug Mode	1.4.7
Interlude: Database Programming	1.5
SQL and SQLite Basics	1.5.1
Creating Tables	1.5.2
Inserting Data	1.5.3
Searching	1.5.4
Updating and Deleting	1.5.5
Working with Multiple Tables	1.5.6

SQL Functions	1.5.7
Example Application	1.5.8
SQL Summary	1.5.9
Chapter Summary	1.5.10
Flask Blog App	1.6
Project Structure	1.6.1
Model	1.6.2
Controller	1.6.3
Views	1.6.4
Templates	1.6.5
Sanity Check	1.6.6
User Login	1.6.7
Sessions	1.6.8
Show Posts	1.6.9
Add Posts	1.6.10
Style	1.6.11
Conclusion	1.6.12
Interlude: Debugging in Python	1.7
Workflow	1.7.1
Breakpoints	1.7.2
Post Mortem Debugging	1.7.3
Flask: FlaskTaskr, Part 1 - Quick Start	1.8
Getting Started	1.8.1
Configuration	1.8.2
Database	1.8.3
Controller	1.8.4
Templates and Styles	1.8.5
Sanity Check	1.8.6
Tasks	1.8.7
Add, Update, and Delete Tasks	1.8.8

Tasks Template	1.8.9
Add Tasks Form	1.8.10
Sanity Check	1.8.11
Flask: FlaskTaskr, Part 2 - SQLAlchemy and User Management	1.9
Database Management	1.9.1
User Registration	1.9.2
Authentication	1.9.3
Database Relationships	1.9.4
Managing Sessions	1.9.5
Flask: FlaskTaskr, Part 3 - Error Handling and Testing	1.10
Error Handling	1.10.1
Testing	1.10.2
Getting Started	1.10.3
Interlude: Intro to HTML and CSS	1.11
HTML	1.11.1
CSS	1.11.2
Chrome Developer Tools	1.11.3
Flask: FlaskTaskr, Part 4 - Styles, Test Coverage, and Permissions	1.12
Templates and Styling	1.12.1
Test Coverage	1.12.2
Nose Testing Framework	1.12.3
Permissions	1.12.4
Flask: FlaskTaskr, Part 5 - Blueprints	1.13
What are Blueprints?	1.13.1
Refactoring	1.13.2
Flask: FlaskTaskr, Part 6 - New features and Error Handling	1.14
New Features	1.14.1
Password Hashing	1.14.2
Custom Error Pages	1.14.3
Flask: FlaskTaskr, Part 7 - Deployment	1.15

Deployment	1.15.1
Automated Deployments	1.15.2
Flask: FlaskTaskr, Part 8 - RESTful API	1.16
Building a RESTful API	1.16.1
Interlude: Flask Boilerplate Template and Workflow	1.17
Flask: FlaskTaskr, Part 9: Continuous Integration and Delivery	1.18
Workflow	1.18.1
Continuous Integration Tools	1.18.2
Travis CI Setup	1.18.3
Intermission	1.18.4
Feature Branch Workflow	1.18.5
Fabric	1.18.6
Recap	1.18.7
Conclusion	1.18.8
Flask: Behavior-Driven Development with Behave	1.19
Behavior-Driven Development	1.19.1
Project Setup	1.19.2
Introduction to Behave	1.19.3
Feature Files	1.19.4
First Feature	1.19.5
Second Feature	1.19.6
Third Feature	1.19.7
Conclusion	1.19.8
Update Steps	1.19.9
Interlude: Web Frameworks, Compared	1.20
Overview	1.20.1
web2py: Quick Start	1.21
Installation	1.21.1
Hello World	1.21.2
Deploying on PythonAnywhere	1.21.3

seconds2minutes App	1.21.4
Interlude: APIs	1.22
Retrieving Web Pages	1.22.1
Web Services Defined	1.22.2
Working with XML	1.22.3
Working with JSON	1.22.4
Working with Web Services	1.22.5
My API Films	1.22.6
web2py: Sentiment Analysis	1.23
Sentiment Analysis	1.23.1
Sentiment Analysis Expanded	1.23.2
Movie Suggester	1.23.3
web2py: py2manager	1.24
Setup	1.24.1
Database	1.24.2
URL Routing	1.24.3
Initial Views	1.24.4
Templates	1.24.5
Profile Page	1.24.6
Add Projects	1.24.7
Add Companies	1.24.8
Homepage	1.24.9
More Grids	1.24.10
Notes	1.24.11
Error Handling	1.24.12
Interlude: Web Scraping and Crawling	1.25
Libraries	1.25.1
HackerNews (scrapy.Spider)	1.25.2
Scrapy Shell	1.25.3
Wikipedia (scrapy.Spider)	1.25.4

Socrata (CrawlSpider and Item Pipeline)	1.25.5
Web Interaction	1.25.6
web2py: REST Redux	1.26
Basic REST	1.26.1
Advanced REST	1.26.2
Django: Quick Start	1.27
Installation	1.27.1
Hello, World!	1.27.2
Workflow	1.27.3
Interlude: Introduction to JavaScript and jQuery	1.28
Getting Started	1.28.1
Handling the Event	1.28.2
Updating the DOM	1.28.3
Django Bloggy: A blog app (part one)	1.29
Setup	1.29.1
Model	1.29.2
Setup an App	1.29.3
Django Shell	1.29.4
Unit Tests for Models	1.29.5
Django Admin	1.29.6
Custom Admin View	1.29.7
Templates and Views	1.29.8
Friendly Views	1.29.9
Django Migrations	1.29.10
View Counts	1.29.11
Styles	1.29.12
Popular Posts	1.29.13
Django Bloggy: A blog app (part two)	1.30
Forms	1.30.1
Even Friendlier Views	1.30.2

Stretch Goals	1.30.3
Django Workflow	1.31
Bloggy Redux: Introducing Blongo	1.32
MongoDB	1.32.1
Talking to Mongo	1.32.2
Django Setup	1.32.3
Setup an App	1.32.4
Add Data	1.32.5
Update Project	1.32.6
Test	1.32.7
Test Script	1.32.8
Conclusion	1.32.9
Django: Ecommerce Site	1.33
Rapid Web Development	1.33.1
Prototyping	1.33.2
Project Setup	1.33.3
Landing Page	1.33.4
Bootstrap	1.33.5
About Page	1.33.6
Contact App	1.33.7
User Registration with Stripe	1.33.8
Next Steps	1.33.9
Appendix A: Installing Python	1.34
Check Current Version	1.34.1
Install Python	1.34.2
Verify Install	1.34.3
Appendix B: Supplementary Materials	1.35
Working with FTP	1.35.1
Working with SFTP	1.35.2
Sending and Receiving Email	1.35.3

Real Python Part 2: Web Development with Python by Michael Herman.

Copyright 2016 Real Python. All rights reserved.

Introduction

This is not a reference book. Instead, we've put together a series of tutorials and examples to highlight the power of using Python for web development. The purpose is to open doors, to expose you to the various options available, so that you can decide the path to go down when you are ready to move beyond this course. Whether it's moving on to more advanced materials, becoming more engaged in the Python development community, or building dynamic web applications of your own - the choice is yours.

This course moves fast, focusing more on practical solutions than theory and concepts. Take the time to go through each example. Ask questions. Join a local [Meetup](#) group. Participate in a hackathon. Take advantage of the various online and offline resources available to you. Engage.

Regardless of whether you have past experience with Python or web development, we urge you to approach this course with a beginner's mind. The best way to learn this material is by challenging yourself. Take the examples further. Find errors in our code. And if you run into a problem, use the "**Google-it-first**" approach/algorithm to find a relevant blog post or article to help answer your question. This is how "real" developers solve "real" problems.

By learning through a series of exercises that are challenging, you will screw up at times. Try not to beat yourself up. Instead, learn from your mistakes - and get better.

NOTE: If you do decide to challenge yourself by finding and correcting code errors or areas that you feel lack clarity, please contact us at info@realpython.com so we can update the course. Any feedback is appreciated. This will benefit other readers, and you will receive credit.

Why Python?

Python is a beautiful language. It's easy to learn and fun, and its syntax (the rules) is clear and concise. Python is a popular choice for beginners, yet still powerful enough to back some of the world's most popular products and applications from companies like NASA, Google, IBM, Cisco, Microsoft, Industrial Light & Magic, among others. Whatever the goal, Python's design makes the programming experience feel almost as natural as writing in English.

As you found out in the previous [course](#), Python is used in a number disciplines, making it extremely versatile. Such disciplines include:

1. System administration
2. 3D animation and image editing
3. Scientific computing/data analysis
4. Game development
5. Web development

With regard to web development, Python powers some of the world's most popular sites. Reddit, the Washington Post, Instagram, Quora, Pinterest, Mozilla, Dropbox, Yelp, and YouTube are all [powered by Python](#).

Unlike Ruby, Python offers a plethora of [frameworks](#) from which to choose from, including bottle.py, Flask, CherryPy, Pyramid, Django, and web2py, to name a few. This freedom of choice allows *you* to decide which framework is best for your applications. You can start with a lightweight framework (Flask, bottle.py) to get a project off the ground quickly, adding complexity as your site grows. Such frameworks are great for beginners who wish to learn the nuts and bolts that underlie web frameworks. Or if you're building an enterprise-level application, the higher-level frameworks (Django, web2py) bypass much of the monotony inherent in web development, enabling you to get an application up quickly and efficiently.

Python 2 vs 3

When Python 3 was released back in 2008 there was very slow adoption of the new version and even some backlash from the well established Python community. This has lead to a bit of debate as to which version of Python to use/learn. Without a doubt, *Python 3 is the way forward*. This course will use Python 3 exclusively, but that does not mean Python 2 will be out of your reach. Most code written in 3, with a few minor

tweaks, can also run in 2. Kenneth Love, an active contributor and educator in the Python community as well as a friend of mine, feels the same. Check out his blog post on the subject [here](#).

Who should take this Course?

The ideal reader should have some background in a programming language. If you are completely new to Python, you should consider starting with the original [Real Python](#) course to learn the fundamentals. The examples and tutorials in this course are written with the assumption that you already have basic programming knowledge.

Please be aware that learning both the Python language and web development at the same time will be confusing. Spend at least a week going through the original course before moving on to web development. Combined with this course, you will get up to speed with Python and web development quickly and smoothly.

What's more, this book is built on the same principles of the original Real Python [course](#):

We will cover the commands and techniques used in the vast majority of cases and focus on how to program real-world solutions to problems that ordinary people actually want to solve.

How to use this Course

This course has roughly three sections: *Client-Side Programming*, *Server-Side Programming*, and *Full-Stack Web Development*. Although each is intended to be modular, the chapters build on one another - so working in order is recommended.

Each lesson contains conceptual information and hands-on, practical exercises meant to reinforce the theory and concepts; many chapters also include homework assignments to further reinforce the material and begin preparing you for the next chapter. A number of videos are [included](#) as well, covering many of the exercises and homework assignments.

Learning by Doing

Since the underlying philosophy is learning by doing, do just that; *type* each and every code snippet presented to you.

Do not copy and paste.

You will learn the concepts better and pick up the syntax faster if you type each line of code out yourself. Plus, if you screw up - which will happen over and over again - the simple act of correcting typos will help you learn how to debug your code.

The lessons work as follows: After we present the main theory, you will type out and then run a small program. We will then provide feedback on how the program works, focusing specifically on new concepts presented in the lesson.

Finish all review exercises and give each homework assignment and the larger development projects a try on your own before getting help from outside resources. You may struggle, but that is just part of the process. You will learn better that way. If you get stuck and feel frustrated, take a break. Stretch. Re-adjust your seat. Go for a walk. Eat something. Do a one-armed handstand. Ask for help if you get stuck for more than a few hours. There is no need to waste time. If you continue to work on each chapter for as long as it takes to at least finish the exercises, eventually something will *click* and everything that seemed hard, confusing, and beyond comprehension will suddenly seem easy.

With enough practice, you will learn this material - and hopefully have fun along the way!

Course Repository

Like the first course, this course has an accompanying [repository](#). Broken up by chapter, you can check your code against the code in the repository after you finish each chapter.

NOTE: You can download the course files directly from the repository. Press the 'Download ZIP' link to download the most recent version of the code as a zip archive. **Be sure to download the updated code for each release.**

License

This e-book and course are copyrighted and licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. This means that you are welcome to share this book and use it for any non-commercial purposes so long as the entire book remains intact and unaltered. That being said, if you have received this copy for free and have found it helpful, I would very much appreciate it if you purchased a copy of your [own](#).

The example Python scripts associated with this book should be considered open content. This means that anyone is welcome to use any portion of the code for any purpose.

Conventions

Formatting

Example code:

```
print("Hello world!")
```

Terminal commands:

```
$ python hello-world.py
```

(dollar signs are not part of the command)

File names:

hello-world.py.

Important terms:

Important term: This is an example of what an important term should look like.

NOTES, WARNINGS, and SEE ALSO boxes:

NOTE: This is a note filled in with bacon ipsum text. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

WARNING: This is a warning also filled in with bacon ipsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

SEE ALSO: This is a see also box with more tasty ipsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

Errata

We welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Or did you find an error in the text or code? Did we omit a topic you would love to know more about. Whatever the reason, good or bad, please send in your feedback.

You can find my contact information on the Real Python [website](#). Or submit an issue on the Real Python official support [repository](#). Thank you!

NOTE: The code found in this course has been tested on Mac OS X v. 10.11.6, Windows XP, Windows 7, Linux Mint 17, and Ubuntu 14.04 LTS.

Getting Started

Let's start with the basics...

Python Review

Before we begin, you should already have Python installed on your machine. We will be using Python 3.6 (the latest as of writing) for the majority of this course. If you do not have Python installed, please refer to Appendix A for a basic tutorial.

To get the most out of this course, you must have at least an understanding of the basic building blocks of the Python language:

- Data Types
- Numbers
- Strings
- Lists
- Operators
- Tuples
- Dictionaries
- Loops
- Functions
- Modules
- Booleans

Again, if you are completely new to Python or just need a brief review, please start with the original Real Python [course](#).

Development Environments

Once Python is installed, take some time familiarizing yourself with the three environments in which we will be using Python with: The command line, the Python Shell, and an advanced Text Editor called [Sublime Text](#). If you are already familiar with these environments, and have Sublime installed, you can skip ahead to the lesson on SQLite - "Installing SQLite".

The Command Line

We will be using the command line, or terminal, extensively throughout this course for navigating your computer and running programs. If you've never used the command line before, please familiarize yourself with the following commands:

Windows	Unix	Action
cd	pwd	show the current path
cd DIR_NAME	cd DIR_NAME	move in one directory level
cd ..	cd ..	move out one directory level
dir	ls	output directory contents
cls	clear	clear the screen
del FILE_NAME	rm FILE_NAME	delete a file
md DIR_NAME	mkdir DIR_NAME	create a new directory
rd DIR_NAME	rmdir DIR_NAME	remove a directory

For simplicity, all command line examples use the Unix-style prompt:

```
$ python3 big_snake.py
```

(The dollar sign is not part of the command.)

Windows equivalent:

```
C:\> python3 big_snake.py
```

Tips

1. Stop for a minute. Within the terminal, hit the UP arrow on your keyboard a few times. The terminal saves a history - called the **command history** - of the commands you've entered in the past. You can quickly re-run commands by arrowing through them and pressing Enter when you see the command you want to run again. You can do the same from the Python shell.
2. Tab can be used to auto-complete directory and file names. For example, if you're in a directory that contains another directory called "directory_name", type CD then the letter 'd' and then press Tab - the directory name will be auto-completed. Now just press Enter to change into that directory. If there's more than one directory that starts with a 'd' you will have to type more letters for the auto-complete to kick in.

For example, if you have a directory that contains the folders "directory_name" and "downloads", you'd have to type `cd di` then tab for auto-complete to pick up on the "directory_name" directory. Try this out. Use your new skills to create a directory. Enter that directory. Create two more directories, naming them "directory_name" and "downloads", respectively. Now test Tab auto-complete.

Both of these tips should save you a few thousand keystrokes in the long run. Practice.

Practice

1. Navigate to your "Desktop".
2. Make a directory called "test".
3. Enter the newly created directory.
4. Create a new directory within "test" called "test-two".
5. Move in one directory, and then create a new directory called "test-three".
6. Use `touch` to create a new file - `touch test-file.txt`.
7. Your directory structure should now look like this:

```
└── test
    └── test-two
        └── test-three
            └── test-file.txt
```

8. Move out two directories. Where are you? You should be in the "test" directory, correct?.
9. Move in to "test-three" again and remove the file - `rm test-file.txt`.
10. Navigate all the way back out to your "Desktop", removing each directory along the

way.

11. Be sure to remove the "test" directory as well.

Questions

1. Open a new terminal/command line. Which directory are you in?
2. What's the fastest way to get to your root directory? We did not cover this. Turn to Google if you need help.
3. How do you check to see the directories and files in the current directory?
4. How do you change directories? How do you create directories?

The Python Shell

The Shell can be accessed through the terminal by typing `python3` and then pressing Enter. The Shell is an interactive environment: You type code directly into it, sending the code directly to the Python Interpreter, which then reads and responds to the entered code. The `>>>` symbols indicate that you're working within the Shell.

Try accessing the Shell through the terminal and print something out:

```
$ python3
>>> phrase = "The bigger the snake, the bigger the prey"
>>> print(phrase)
The bigger the snake, the bigger the prey
```

To exit the Shell from the terminal, press CTRL-Z + Enter within Windows, or CTRL-D within Unix. You can also just type `exit()` then press enter:

```
>>> exit()
$
```

The Shell gives you an immediate response to the code you enter. It's great for testing, but you can really only run one statement at a time. To write longer scripts, we will be using a text editor called Sublime Text.

Sublime Text

Again, for much of this course, we will be using a basic yet powerful text editor built for writing source code called Sublime Text. Like Python, it's cross-compatible with many operating systems. Sublime works well with Python and offers excellent syntax

highlighting - applying colors to certain parts of programs such as comments, keywords, and variables based on the Python syntax - making the code more readable.

You can download the latest versions for Windows and Unix [here](#). It's free to try for as long as you like, although it will bug you a number of times a day to purchase a license.

Once downloaded, go ahead and open the editor. At first glance it may look just like any other text editor. It's not. It can be used like that, but you can take advantage of all the powerful built-in features it possesses along with the various packages used to extend its functionality. There's way more to it than what meets the eye. But slow down. We don't want to move too fast, so we'll start with the basics first and use it as a text editor. Don't worry - you'll have plenty of time to learn all the cool features soon enough.

There are plenty of other free text editors that you can use such as Notepad++ for Windows, TextWrangler for Mac, and the excellent, cross-platform editors Atom and gedit. If you are familiar with a more powerful editor or IDE, feel free to use it. However, we can't stress enough that you must be familiar with it *before* starting this course. Do not try to take this course while also learning how to learn an IDE. You'll just add another layer of complexity to the entire process.

Really, almost any text editor will do, however all examples in this course will be completed using Sublime Text.

Python files must be indented with four spaces, not tabs. Most editors will allow you to change the indentation to spaces. Go ahead and do this within your editor. Jump to this [link](#) to see how to do this in Sublime. The example shows `tab_size` set to 2 spaces, but be sure to set it to `4` for easy python coding.

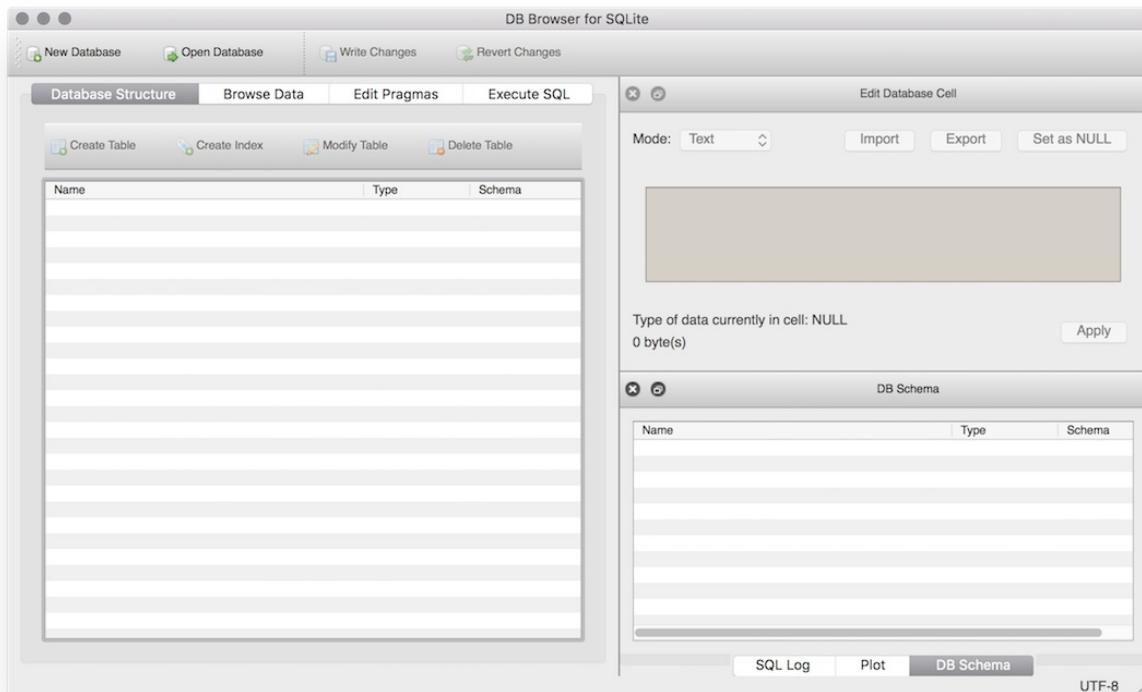
WARNING: Never use a word processor like Microsoft Word as your text editor, because text editors, unlike word processors, just display raw, plain text.

Homework

- Create a directory using your terminal within your "Documents" or "My Documents" directory called "RealPython". All the code from your exercises and homework assignments will be saved in this directory.
- To speed up your workflow, you should create an alias to this directory so you can access it much quicker. To learn more about setting up aliases on a Mac, please read [this](#) article.

Installing SQLite

In a few chapters we will begin working with relational databases. You will be using the SQLite database because it's simple to set up and great for beginners who need to learn the SQL syntax. Python includes the SQLite library. We just need to install the SQLite Database Browser:



Regardless of the operating system, you can download the SQLite Database Browser from [here](#). Installation for Windows and Mac OS X environments are relatively the same. As for Linux, installation is again dependent upon which Linux flavor you are using.

Homework

- Read [Learn SQL Or Create A Simple Database With SQLite Database Browser](#) to learn the basics of the SQLite Database Browser.

pip

pip is a package management system used for installing and managing Python packages. In other words, it's used for installing third-party packages and libraries that other Python developers create. For example, if you wanted to work with a third party API like Twitter or Google Maps, you can save a lot of time by using a pre-built package, rather than building all of the functionality yourself from scratch.

To install a package, run:

```
$ pip3 install numpy
```

To download a specific version of a package:

```
$ pip3 install numpy==1.10.1
```

To uninstall a package:

```
$ pip3 uninstall numpy
```

NOTE: If you are in a Unix environment you may need to use `sudo` before each command in order to execute it as the root user - `sudo pip3 install numpy`. You will then have to enter your root password to install.

virtual environment

It's common practice to use a *virtual environment* for your various projects, which is used to create isolated Python environments (also called "sandboxes"). Essentially, when you work on one project, it will not affect any of your other projects.

It's *absolutely* essential to work in a virtual environment so that you can keep all of your Python versions, libraries, packages, and dependencies separate from one another.

Some examples:

1. Simultaneous applications you work on may have different dependency requirements for one particular package. One application may need version 1.3.1 of package X, while another could require version 1.2.3. Without virtual environment, you would not be able to access each version concurrently.
2. You have one application written in Python 2.7 and another written in Python 3.6. Using virtual environment to separate the development environments, both applications can reside on the same machine without creating conflicts.
3. One project uses Django version 1.2 while another uses version 1.8. Again, virtual environment allows you to have both versions installed in isolated environments so they don't affect each other.

Python will work fine without virtual environment. But if you start working on a number of projects with a number of different libraries installed, you will find virtual environment an absolute necessity. Once you understand the concept behind it, virtual environment is easy to use. It will save you time (and possibly prevent a huge headache) in the long run.

Let's practice creating a virtual environment:

1. Navigate to your "RealPython" directory.
2. Create a new directory called "real-python-test".
3. Navigate into the new directory, then run the following command to set up a virtual environment within that directory:

```
$ python3 -m venv env
```

This created a new directory, "env", and set up a virtual environment within that directory. Now you have a completely isolated environment, free from any previously installed packages.

4. Now you just need to activate the virtual environment, enabling the isolated work environment:

Unix:

```
$ source env/bin/activate
```

Windows:

```
$ env\scripts\activate
```

This changes the context of your terminal in that folder, "real-python-test", which acts as the root of your system. You can tell when you're working in a virtual environment by the directory surrounded by parentheses to the left of the path in your command-line:

```
$ source env/bin/activate  
(env)$
```

When you're done working, you can then exit the virtual environment using the `deactivate` command. And when you're ready to develop again, simply navigate back to that same directory and reactivate the virtual environment - `source env/bin/activate`.

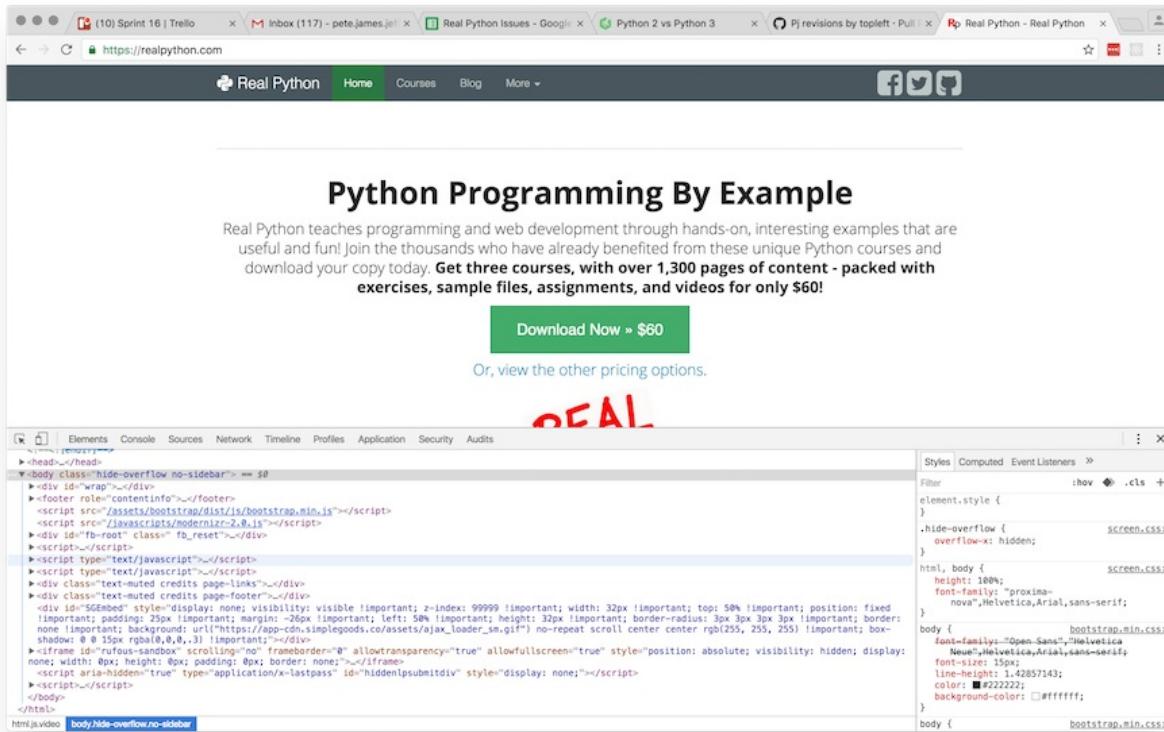
Go ahead and deactivate the virtual environment.

NOTE: You may see a lot of tutorials out there use the command `pip` and not `pip3` like we used above. When working inside of an activated virtual environment, the `pip` and `pip3` commands are interchangeable.

Every time you create a new project, install a new virtual environment.

Web Browsers

We will be using Chrome for this course, because we will make use of a powerful, pre-installed set of tools for web developers called Chrome Developer Tools. These tools allow you to inspect and analyze various elements that make up a web page (or app). You can view HTTP requests, test CSS style changes, and debug code, among others.



NOTE: Use the keyboard, save time! On a mac you can hit `Cmd + Opt + I` to open the developers tools when inside a Chrome browser window. On Windows and Linux use `Ctrl + Shift + I`.

Version Control

Finally, to complete our development environment set up, we need to install a version control system. Such systems allow you to save different "versions" of a project. Over time, as your code and your project becomes bigger, it may become necessary to backtrack to an earlier "version" to undo changes if a giant error occurs. It will happen. We will be using Git for version control and Github for remotely hosting our code.

Setting up Git

It's common practice to put projects under version control before you start developing by creating storage areas called repositories (or repos).

This is an essential step in the development process. Again, such systems allow you to easily track changes when your codebase is updated as well as reverted (or rolled back) to earlier versions of your codebase in case of an error.

Take the time to learn how to use a version control system. It is a *required* skill for web developers to have.

Start by downloading [Git](#), if you don't already have it. Make sure to download the version appropriate for your system. If you've never installed Git before you need to set your global first name, last name, and email address. Open the terminal in Unix or the Git Bash Shell in Windows (Start > All Programs > Git > Git Bash), then enter the following commands:

```
$ git config --global user.name "FIRST_NAME LAST_NAME"  
$ git config --global user.email "MY_NAME@example.com"
```

We highly recommend reading Chapter 2 and 3 of the [Git Book](#). *Do not worry if you don't understand everything, as long as you grasp the high-level concepts and workflow. You'll learn better by doing, anyway.*

Introduction to Github

Github is related to Git, but it is distinct. While you use Git to create and maintain local repositories, Github is a collaborative, social network used to remotely host Git repositories so-

- Your projects are safe from potential damages to your local machine,
- You can show off your projects to potential employers (think of Github as your online resume), and
- Other users can collaborate on your project.

Get used to Github. You'll be using it a lot. It's the most popular place for programmers to host their projects.

1. Sign up for a new account on [Github](#). It's free!
2. Click the "New Repository" button on the main page. Or, simply follow [this](#) link.
3. Name the repository "real-python-test", then in the description add the following text, "Real Python test!".
4. Click "Create Repository".

Congrats! You just set up a new repository on Github. Leave that page up while we create a local repository using Git.

Create a Local Repo

Go to your "real-python-test" folder to initialize a local repo (make sure to activate your virtual environment first):

```
$ git init
```

Next add a file called *README.md*. It's convention to have such a file, stating the purpose of your project, so that when added to Github, other users can get a quick sense of what your project is all about.

```
$ touch README.md
```

Open that file in Sublime and just type "Hello, world! This is my first PUSH to Github." Save the file. Now let's add the file to your local repo. First we need to take a "snapshot" of the project in it's current state:

```
$ git add -A
```

This essentially adds all files that have either been created, updated, or deleted to a place called "staging". Don't worry about what that means; just know that your project is not part of the local repo yet.

To add the project to your repo, you need to run the following command:

```
$ git commit -m "init"
```

Sweet! Now your project has been committed (or added) to your local repo. Let's add it to Github now.

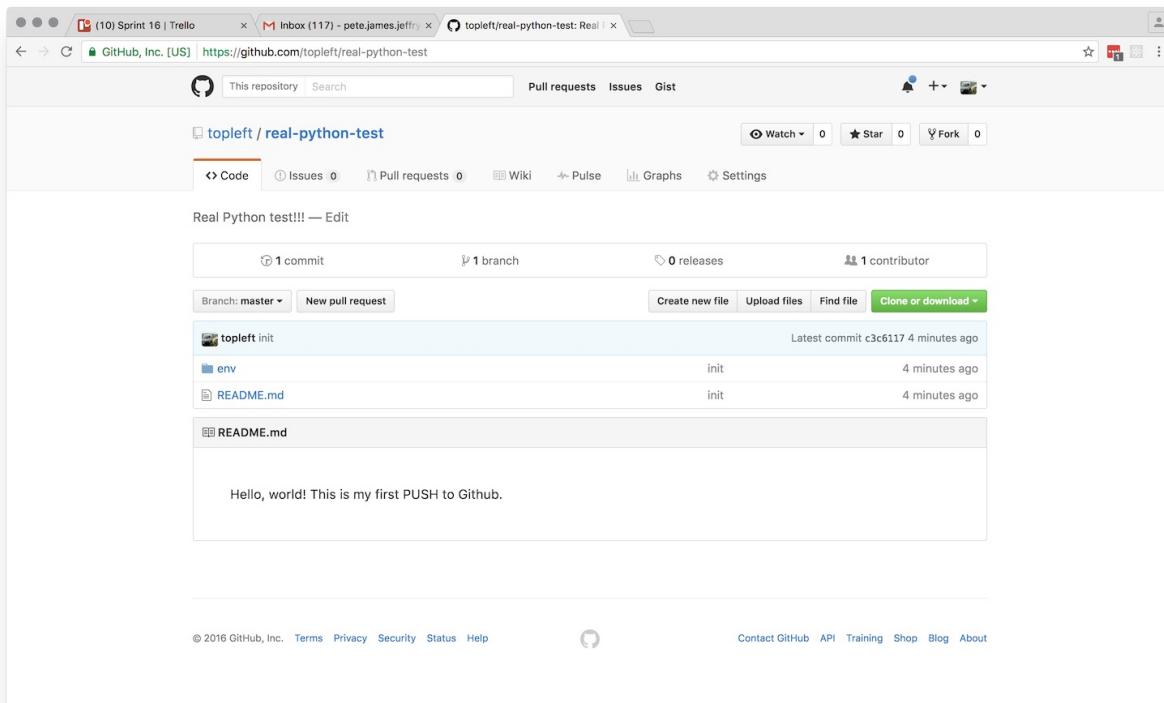
Add a link to your remote repository. Return to Github. Copy the command to add your remote repo, then paste it into your terminal:

```
$ git remote add origin https://github.com/<YOUR-USERNAME>/real-python-test.git
```

Finally, PUSH the local repo to Github:

```
$ git push origin master
```

Open your browser and refresh the Github page. You should now see the files from your local repository on Github.



That's it!

Review

Let's review.

When starting a new repository, you need to follow these steps:

Add the repo on Github. Run the following commands in your local directory:

```
$ git init
$ touch README.md
$ git add -A
$ git commit -m "some message"
$ git remote add origin https://github.com/<YOUR-USERNAME>/<YOUR-REPO-NAME>.git
$ git push origin master
```

Again, this creates the necessary files and pushes them to the remote repository on Github.

Next, after your repo has been created locally and remotely - and you completed your first PUSH - you can follow this similar workflow to PUSH as needed:

```
$ git add -A
$ git commit -am 'message'
$ git push origin master
```

NOTE: The string within the commit command should be replaced each time with a description of the changes made in that commit.

That's it. With regard to Git, it's essential that you know how to:

- Create a local and remote repo
- Add and commit
- Push your local copy to Github

We'll get into more specifics as we progress through the course.

Git Workflow

Let's review. You have one directory on your local computer called "RealPython". The goal is to add all of our projects to that folder, and each of those individual folders will have a separate Git repo that needs to also stay in sync with Github.

To do so, each time you make any *major* changes to a specific project, commit them locally and then PUSH them to Github:

```
$ git add -A  
$ git commit -am "some message goes here about the commit"  
$ git push origin master
```

Simple, right?

Next add a `.gitignore` file (no file extension!), which is a file that you can use to specify files you wish to ignore or keep out of your repository, both local and remote. What files should you keep out? If it's clutter (such as a `.pyc` file) or a secret (such as an API key), then keep it out of your public repo on Github.

Please read more about `.gitignore` [here](#). For now, add the files found [here](#).

Your Setup

1. We have one directory, "RealPython", that will contain all of your projects.
2. Each of those project directories will contain a separate virtual environment, Git repo, and `.gitignore` file.
3. Finally, we want to set up a `requirements.txt` file for *each* virtual environment. This file contains a list of packages you've installed via pip. This is meant to be used by other developers to recreate the same development environment. To do this, run the following command when your virtual environment is activated:

```
$ pip freeze > requirements.txt
```

NOTE: Although this is optional, we highly recommend setting up a RSA key for use with Github so you don't have to enter your password each time you push code to Github. Follow the instructions [here](#).

Homework

- Please read more about the basic Git commands [here](#).

Interlude: Modern Web Development

Even though this is a Python course, we will be using a number of other tools and technologies such as HTML, CSS, JavaScript/jQuery, AJAX, and SQL, to name a few. You will be learning all the technologies, on the front-end and back-end, which work in conjunction with Python web frameworks, needed to create a full-stack web application.

Much of your learning will start from the ground up, so you will develop a deeper understanding of web frameworks, allowing for quicker and more flexible web development.

Before we start, let's look at an overview of modern web development.

Front-end, Back-end, and Middleware

Modern web applications are made up of three parts or layers, each of which encompasses a specific set of technologies, that are constantly working together to deliver the desired results.

1. **Front-end:** The presentation layer. It's what the end user sees when interacting with a web application. HTML defines the structure, while CSS provides a pretty facade, and JavaScript provides user interaction. Essentially, this is what the end user sees when interacting with your web application through a web browser. The front-end is reliant on the application logic and data sources provided by the middleware and back-end.
2. **Middleware:** This layer relays information between the front and back-end, in order to:
 - Process HTTP requests and responses;
 - Connect to the server;
 - Interact with APIs; and
 - Manage URL routing, authentication, sessions, and cookies.
3. **Back-end:** This is where data is stored, analyzed, and processed. Languages such as Python, PHP, and Ruby communicate back and forth with the database, web service, or other data source to produce the end-user's requested data.

NOTE: Don't worry if you don't understand all of this. We will be discussing each of these technologies and how they fit into the whole throughout the course.

Developers adept in web architecture and in programming languages like Python, have traditionally worked on the back-end and middleware layers, while designers and those with HTML, CSS, and JavaScript skills, have focused on the front-end. These roles are becoming less and less defined, especially in start-ups. Traditional back-end developers are now also handling much of the front-end work. This is due to both a lack of quality designers and the emergence of front-end CSS frameworks, like Bootstrap and Foundation, which have significantly sped up the design process.

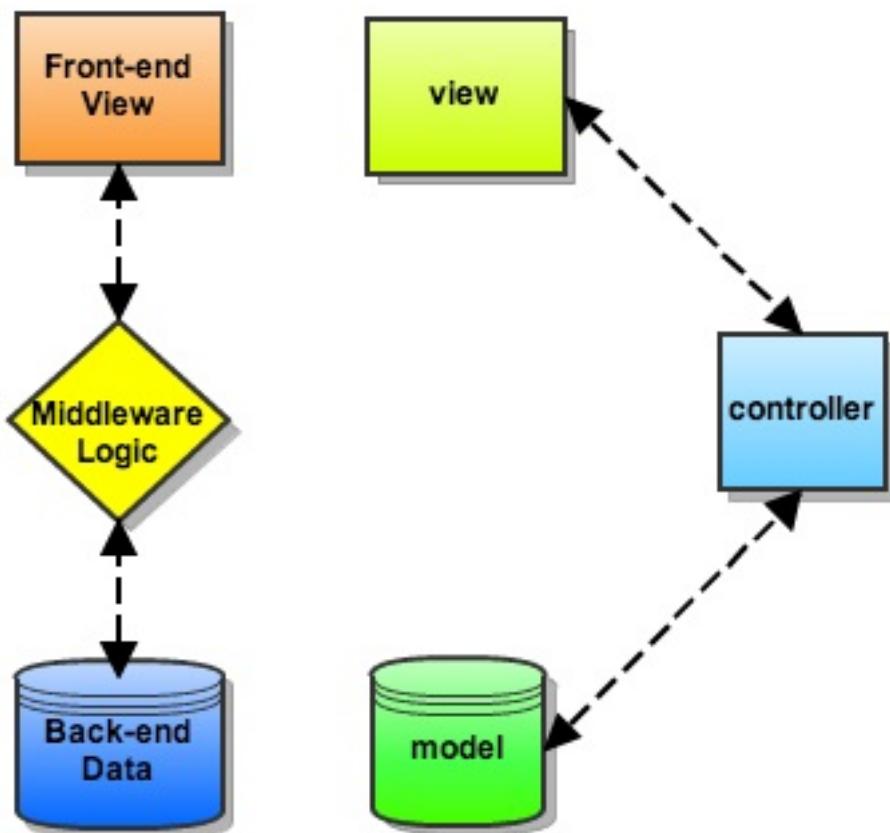
Model-View-Controller (MVC)

Web frameworks reside just above those three layers, abstracting away much of the processes that occur within each. There are pros and cons associated with this: It's great for experienced web developers who understand the automation (or *magic!*) behind the scenes.

Web frameworks simplify web development, by handling much of the superfluous, repetitious tasks.

This can be very confusing for a beginner, though - which reinforces the need to start slow and work our way up.

Frameworks also separate the presentation (view) from the application logic (controller) and the underlying data (model) in what's commonly referred to as the Model-View-Controller architecture pattern. While the front-end, back-end, and middleware layers operate linearly, MVC *generally* operates in a triangular pattern:



We'll be covering this pattern numerous times throughout the remainder of this course. Let's get to it!

Homework

- Read about the differences between a website (static) and a web application (dynamic) [here](#).
- Want to learn more about MVC? Read [Model-View-Controller \(MVC\) Explained -- With Legos](#)

Flask: Quick Start

This chapter introduces you to the Flask web framework.



Overview

[Flask](#) grew from an elaborate [April fool's joke](#) that mocked single file, micro frameworks (most notably [bottle.py](#)) in 2010 into one of the most popular Python web frameworks in use today. Small yet powerful, you can build your application from a single file, and, as it grows, organically develop components to add functionality and complexity.

Installation

1. Within your "RealPython" directory create a new directly called "flask-hello-world".
2. Navigate into the directory and then create and activate a new virtual environment.
3. Now install Flask:

```
$ pip install flask
```

Hello World

Because it's a well established convention, we'll start with a quick "Hello World" example in Flask. This serves two purposes:

1. To ensure that your setup is correct, and
2. To get you accustomed to the workflow and conventions used throughout the rest of the course.

This app will be contained entirely within a single file, *app.py*. Yes, it's that easy! Open Sublime and within a new file add the following code:

```
# ---- Flask Hello World ---- #

# import the Flask class from the flask package
from flask import Flask

# create the application object
app = Flask(__name__)

# use the decorator pattern to
# link the view function to a url
@app.route("/")
@app.route("/hello")

# define the view using a function, which returns a string
def hello_world():
    return "Hello, World!"

# start the development server using the run() method
if __name__ == "__main__":
    app.run()
```

Save the file as *app.py* and run it:

```
$ python app.py
```

This launches the development server that's listening on port 5000. Open a web browser and navigate to <http://127.0.0.1:5000/>. You should see the "Hello, World!" greeting.

NOTE: <http://127.0.0.1:5000> and <http://localhost:5000> are equivalent. So when you run Flask locally, you can point your browser to either URL. Both will work.

Test out the URL <http://127.0.0.1:5000/hello> as well. Once done, back in the terminal press CTRL-C to stop the server.

What's going on here?

Let's first look at the code without the view function:

```
from flask import Flask

app = Flask(__name__)

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

1. We imported the `Flask` class from the `flask` library in order to create our web app.
2. Next, an instance of the `Flask` class was created and assigned to the variable `app`.
3. Finally, we used the `run()` method to run the app locally.

NOTE: Behind the scenes, the `__name__` variable was set equal to `"__main__"`, indicating that we're running the statements in the current file (as a module) rather than importing it. This conditional ensures that the app will only run if *this* file is called from the command line (more on this later). Also, check out [this](#) link for more information.

Now for the function:

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

1. Here we applied two **decorators** - `@app.route("/")` and `@app.route("/hello")` to the `hello_world()` function. These **decorators** are used to define routes. In other words, we created two routes - `/` and `/hello` - which are bound to our main url, <http://127.0.0.1:5000>. Thus, we are able to access the function by navigating to either <http://127.0.0.1:5000> or <http://127.0.0.1:5000/hello>.
2. The function simply returned the string "Hello, World!".

SEE ALSO: Want to learn more about decorators? Check out these two blog posts - [Primer on Python Decorators](#) and [Understanding Python Decorators in 12 Easy Steps!](#).

Finally, we need to commit to Github.

```
$ git add .
$ git commit -am "flask-hello-world"
$ git push origin master
```

From now on, I will remind you by just telling you to commit and PUSH to Github and the end of each chapter. Make sure you remember to do it after each lesson, though!

App Flow

Before moving on, let's pause for a minute and talk about the flow of the application from the perspective of an end user:

The end user (you) requests to view a web page at the URL <http://127.0.0.1:5000/hello>.

The controller then handles the request determining what should be displayed back, based on the URL that was entered, the functionality directly below the route definition, as well as the requested HTTP method (more on this later):

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

Since the function directly below the route definition simply returns the text "Hello, World!", the controller can render the HTML as soon as the route handler is hit. In some cases, depending on the request, the controller may need to grab data from a database and perform necessary calculations on said data, before rendering the HTML. The HTML is rendered and displayed to the end user:

```
return "Hello, World!"
```

Make sense? Don't worry if it's still a bit confusing. Just keep this flow in mind as you develop more apps throughout this course. It will click soon enough.

Dynamic Routes

Thus far we've only looked at static routes. Let's create something dynamic.

Add a new route to `app.py`:

```
# dynamic route
@app.route("/test")
def search():
    return "Hello"
```

Test it out.

NOTE: Whenever you update the code in your text editor, you must kill (CTRL+C) and restart your server from the command line to see the changes in your browser.

Now to make it dynamic first update the route to take a query parameter:

```
@app.route("/test/<search_query>")
```

Next, update the function so that it takes the query parameter as an argument that simply returns it:

```
def search(search_query):
    return search_query
```

Navigate to <http://localhost:5000/test/hi>. You should see "hi" on the page. Test it out with some different URL parameters.

URLs are generally converted to a string, regardless of the parameter. For example, in the URL <http://localhost:5000/test/101>, the parameter of 101 is converted into a string. What if we wanted it treated as an integer though? Well, we can change how parameters are treated with [converters](#).

Flask converters:

- `<value>` is treated as unicode (string)
- `<int:value>` is treated as an integer
- `<float:value>` is treated as a floating point

- `<path:some/great/path/>` is treated as a path

Test this out. Create three new route handlers:

```
@app.route("/integer/<int:value>")  
def int_type(value):  
    print(value + 1)  
    return "correct"  
  
@app.route("/float/<float:value>")  
def float_type(value):  
    print(value + 1)  
    return "correct"  
  
# dynamic route that accepts slashes  
@app.route("/path/<path:value>")  
def path_type(value):  
    print(value)  
    return "correct"
```

First test - <http://localhost:5000/integer/1>. You should see the value 2 in your terminal. Then try a string and a float. You should see a 404 error for each.

Second test - <http://localhost:5000/float/1.1>. You should see the value 2.1 in your terminal. Both a string and an integer will return 404 errors.

Third test - <http://localhost:5000/path/just/a/random/path>. You should see the path in your terminal. You can use both integers and floats as well, but they will be converted to unicode.

Response Object

Notice that in the response object (`return search_query`) we are only supplying text. This object is actually a tuple that can take two more elements - the HTTP status code and a dictionary of headers, both of which are optional:

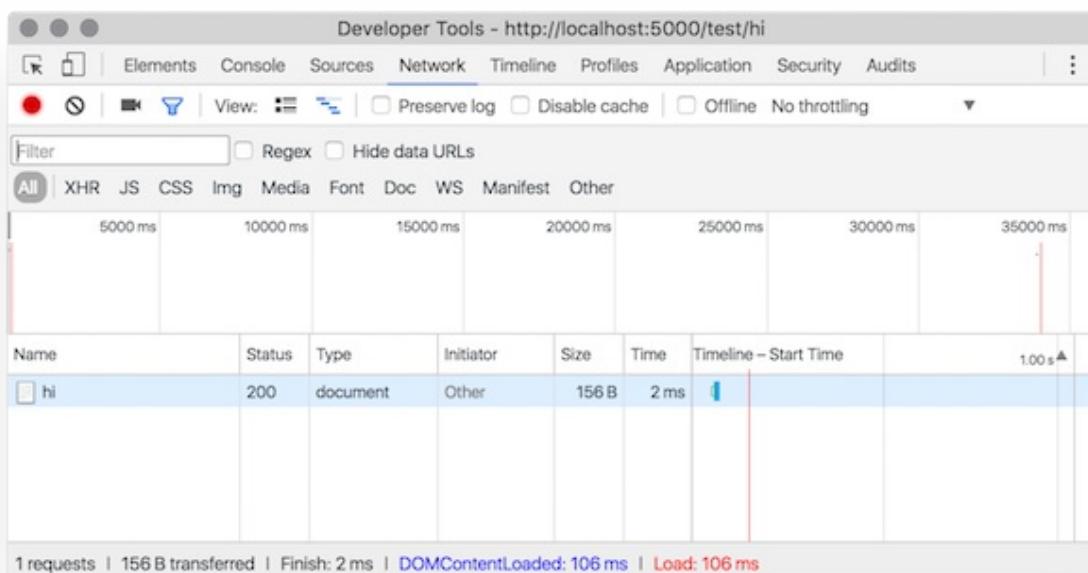
```
(response, status, headers)
```

If you do not explicitly define these, Flask will automatically assign a Status Code of 200 and a header where the Content-Type is "document" when a string is returned.

For example, navigate to this URL again, <http://localhost:5000/test/hi>. Next open Chrome Developer Tools: Right click anywhere on the page, scroll down to "Inspect Element.", click the "Network" tab within Developer Tools.

Refresh your page.

Watch Developer Tools:



Notice the 200 Status Code and Content-Type. That was the expected behavior.

Let's add a new view:

```
@app.route("/name/<name>")  
def index(name):  
    if name.lower() == "michael" :  
        return "Hello, {}".format(name), 200  
    else :  
        return "Not Found", 404
```

Here, the route can take an optional parameter of name, and the response object is dependent on the assigned value. We also explicitly assigned a status code. With Developer Tools open, try the URL <http://localhost:5000/name/michael>. Watch the response. Then remove "michael" and try any other parameter. You should see the 404 status code. Finally, test the first URL out again, but this time, update the response object to:

```
return "Hello, {}".format(name)
```

Same response as before, right? 200 status code and Content-Type of "document". Flask is smart: It inferred the correct Status Code and Content-Type based on the response type. Magic!

NOTE: Although, the Status Code can be implicitly generated by Flask, it's a common convention to explicitly define it for RESTful APIs since client-side behavior is often dependent on the status code returned. In other words, instead of letting Flask guess the status code, define it yourself to make certain that it is correct. Check out more on status codes [here](#).

Debug Mode

Flask provides helpful error messages and prints stack traces directly in the browser, making debugging much easier. To enable these features along with [automatic reload](#) simply add the following to the `app.py` file:

```
app.config["DEBUG"] = True
```

The code should now look like this:

```
# import the Flask class from the flask module
from flask import Flask

# create the application object
app = Flask(__name__)

# error handling
app.config["DEBUG"] = True

# use the decorator pattern to
# link the view function to a url
@app.route("/")
@app.route("/hello")

# define the view using a function, which returns a string
def hello_world():
    return "Hello, World!"

# dynamic route
@app.route("/test/<search_query>")
def search(search_query):
    return search_query

# dynamic route with explicit status codes
@app.route("/name/<name>")
def index(name):
    if name.lower() == "michael" :
        return "Hello, {}".format(name)
    else :
        return "Not Found", 404

# start the development server using the run() method
if __name__ == "__main__":
    app.run()
```

After you save your file, manually restart your server to start seeing the automatic reloads. Check your terminal, you should see:

```
Detected change in 'some/file/path', reloading
* Restarting with stat
```

Essentially, any time you make changes to the code and save, they will be auto loaded. You do not have to restart your server to refresh the changes; you just need to refresh your browser.

Edit the string returned by the `hello_world()` function to:

```
return "Hello, World!?!?!?"
```

Save your code. Refresh the browser. You should see the new string.

We'll look at error handling in a later chapter. Until then, make sure you commit and PUSH your code to Github.

Kill the server, then deactivate your virtual environment.

SEE ALSO: Want to use a different debugger? See [Working with Debuggers](#).

Interlude: Database Programming

Before moving on to a more advanced Flask application, let's look at database programming.

Nearly every web application has to store data. Without data most web applications would provide little, if any, value. Think of your favorite web app. Now imagine if it contained no data. Take Twitter for example: Without data (in the form of Tweets), Twitter would be relatively useless.

Hence the need to store data in some sort of meaningful way.

One of the most common methods of storing (or persisting) information is to use a relational database. In this chapter, we'll look at the basics of relational databases as well as SQL (Structured Query Language).

Before we start, let's get some terminology out of the way.

Databases

Databases help organize and store data. It's like a digital filing cabinet, and just like a filing cabinet, we can retrieve, add, update, and/or remove data from it. Sticking with that metaphor, databases contain tables, which are like file folders, storing similar information. While folders contain individual documents, database tables contain rows of data.

SQL and SQLite Basics

A database is a structured set of data. Besides flat files, the most popular types of databases are relational databases. These organize information into tables, similar to a basic spreadsheet, which are uniquely identified by their name. Each table is comprised of columns called fields and rows called records (or data).

Here is a sample table called "Employees":

EmpNo	EmpName	DeptNo	DeptName
111	Michael	10	Development
112	Fletcher	20	Sales
113	Jeremy	10	Development
114	Carol	20	Sales
115	Evan	20	Sales

Records from one table can be linked to records in another table to create relationships. More on this later.

Most relational databases use the SQL language to communicate with the database. SQL is a fairly easy language to learn, and one worth learning. The goal here is to provide you with a high-level overview of SQL to get you started. To achieve the goals of the course, you need to understand the four basic SQL commands: SELECT, INSERT, UPDATE, and DELETE.

Command	Action
SELECT	retrieves data from the database
INSERT	inserts data into the database
UPDATE	updates data from the database
DELETE	deletes data from the database

Although SQL is a simple language, you will find an even easier way to interact with databases called Object Relational Mapping, which will be covered in future chapters. In essence, instead of working with SQL directly, you work with Python objects, which

many Python programmers are more comfortable with, that abstract out the SQL language. We'll cover these methods in later chapters. For now, we'll cover SQL, as it's important to understand how SQL works for when you have to troubleshoot or conduct difficult queries that require SQL.

Numerous libraries and modules are available for connecting to relational database management systems. Such systems include SQLite, MySQL, PostgreSQL, Microsoft Access, SQL Server, and Oracle. Since the language is relatively the same across these systems, choosing the one which best suits the needs of your application depends on the application's current and expected size. In this chapter, we will focus on SQLite, which is ideal for simple applications.

[SQLite](#) is great. It gives you most of the database structure of the larger, more powerful relational database systems without having to actually use a server. Again, it is ideal for simple applications as well as for testing out code. Lightweight and fast, SQLite requires little administration. It's also already included in the Python standard library. Thus, you can literally start creating and accessing databases without downloading any additional dependencies.

Homework

- Spend thirty minutes reading more about the basic SQL commands highlighted above from the official [SQLite documentation](#). If you have time, check out W3schools.com's [Basic SQL Tutorial](#) as well. This will set the basic ground work for the rest of the chapter.
- Also, if you have access to the first Real Python [course](#), go through chapter 13 again.

Creating Tables

Let's begin. Make sure you've created a "sql" directory within then "RealPython" directory. Create and activate your virtual environment as well as a Git repo.

Use the `Create Table` statement to create a new table. Here is the basic format:

```
create table table_name
(column1 data_type,
 column2 data_type,
 column3 data_type);
```

Let's create a basic Python script to do this:

```
# Create a SQLite3 database and table

# import the sqlite3 library
import sqlite3

# create a new database if the database doesn't already exist
conn = sqlite3.connect("new.db")

# get a cursor object used to execute SQL commands
cursor = conn.cursor()

# create a table
cursor.execute("""CREATE TABLE population
                (city TEXT, state TEXT, population INT)
                """)

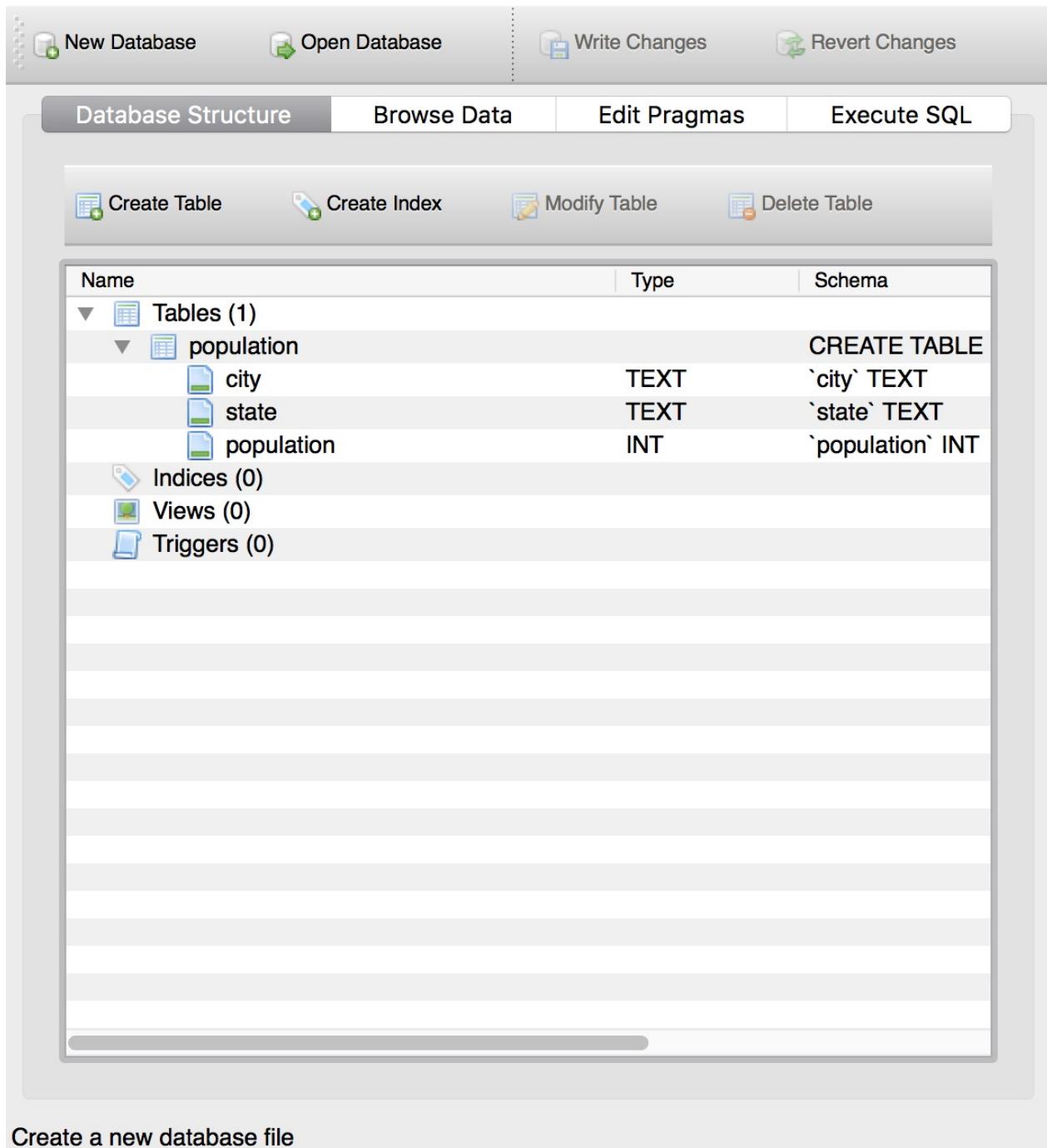
# close the database connection
conn.close()
```

Save the file as `01_sql.py`. Run the file from your terminal:

```
$ python 01_sql.py
```

As long as error wasn't thrown, the database and table were created inside a new file called `new.db`. You can verify this by launching the SQLite Database Browser and then opening up the newly created database, which will be located in the same directory

where you saved the file. Under the "Database Structure" tab you should see the "population" table. You can then expand the table and see the "city", "state", and "population" fields (also called table columns):



Name	Type	Schema
Tables (1)		
population	CREATE TABLE	
city	TEXT	`city` TEXT
state	TEXT	`state` TEXT
population	INT	`population` INT
Indices (0)		
Views (0)		
Triggers (0)		

Create a new database file

So what exactly did we do here?

1. We imported the `sqlite3` library to communicate with SQLite.
2. Next, a new database named `new.db` was created. (This same command is also used to connect to an existing database. Since a database didn't exist in this case, one was created for us.)
3. We then created a cursor object, which lets us execute a SQL query or command

against data within the database.

4. Finally, we created a table named "population" using the SQL statement `CREATE TABLE` that has two text fields, "city" and "state", and one integer field, "population".

NOTE: You can also use the `":memory:"` string to create a database in memory only:

```
conn = sqlite3.connect(":memory:")
```

Keep in mind, though, that as soon as you close the connection the database will disappear.

No matter what you are trying to accomplish, you will usually follow this basic workflow:

1. Create a database connection
2. Get a cursor
3. Execute your SQL query
4. Close the connection

What happens next depends on your end goal. You may insert new data (INSERT), modify (UPDATE) or delete (DELETE) current data, or simply extract data in order to output it to the screen or conduct analysis (SELECT). Go back and look at the SQL statements, from the beginning of the chapter, for a basic review.

NOTE: You can delete a table by using the `DROP TABLE` command plus the table name you'd like to drop - i.e., `DROP TABLE table_name`. This of course deletes the table and all the data associated with that table. **Use with caution.**

Don't forget to commit to Git before moving on!

Homework

- Create a new database called "cars", and add a table called "inventory" that includes the following fields: "Make", "Model", and "Quantity". Don't forget to include the proper data-types.

Inserting Data

Now that we have a table created, we can populate it with some actual data by adding new rows to the table:

```
# INSERT Command

# import the sqlite3 library
import sqlite3

# create the connection object
conn = sqlite3.connect("new.db")

# get a cursor object used to execute SQL commands
cursor = conn.cursor()

# insert data
cursor.execute("INSERT INTO population VALUES('New York City', \
    'NY', 8400000)")
cursor.execute("INSERT INTO population VALUES('San Francisco', \
    'CA', 800000)")

# commit the changes
conn.commit()

# close the database connection
conn.close()
```

Save the file as *02_sql.py* and then run it. Again, if you did not receive an error, then you can *assume* the code ran correctly. Open up the SQLite Database Browser again to ensure that the data was added. After you load the database, click the second tab, "browse data", and you should see the new values that were inserted.

1. As in the example from the previous lesson, we imported the `sqlite3` library, established the database connection, and created the cursor object.
2. We then used the `INSERT INTO` SQL command to insert data into the "population" table. Note how each item (except the integers) has single quotes around it, while the entire statement is enclosed in double quotes. Many relational databases only allow objects to be enclosed in single quotes. This can get a bit more complicated when you have items that include single quotes in them. There is a workaround, though - the `executemany()` method which you will see in the next example.

3. The `commit()` method executes the SQL statements and inserts the data into the table. Anytime you make changes to a table via the INSERT, UPDATE, or DELETE commands, you need to run the `commit()` method before you close the database connection. Otherwise, the values will only persist temporarily in memory.

That being said, if you rewrite your script using the `with` keyword, your changes will automatically be saved without having to use the `commit()` method, making your code more compact.

Let's look at the above code re-written using the `with` keyword:

```
import sqlite3
with sqlite3.connect("new.db") as connection:
    c = connection.cursor()
    c.execute("INSERT INTO population VALUES('New York City', \
        'NY', 8400000)")
    c.execute("INSERT INTO population VALUES('San Francisco', \
        'CA', 8000000)")
```

Using the `executemany()` method

If you need to run a number of the same SQL statements at once you can use the `executemany()` method to save time and eliminate unnecessary code. This can be helpful when initially populating a database with data.

```
# executemany() method

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # insert multiple records using a tuple
    cities = [
        ('Boston', 'MA', 6000000),
        ('Chicago', 'IL', 2700000),
        ('Houston', 'TX', 2100000),
        ('Phoenix', 'AZ', 1500000)
    ]

    # insert data into table
    c.executemany('INSERT INTO population VALUES(?, ?, ?)', cities)
```

Save the file as `03_sql.py` then run it. Double check your work in the SQLite Database Browser that the values were added.

In this example, instead of using string substitution, we used parameterized statements (the question marks). Parametrized statements should always be used when communicating with a SQL database due to potential SQL injections that could occur from using string substitutions.

Essentially, a SQL injection is a fancy term for when a user supplies a value that *looks* like SQL code but really causes the SQL statement to behave in unexpected ways. Whether accidental or malicious in intent, the statement fools the database into thinking it's a real SQL statement. In some cases, a SQL injection can reveal sensitive information or even damage or destroy the database. Be careful.

Importing data from a CSV file

In many cases, you may need to insert thousands of records into your database, in which case the data is probably contained within an external [CSV file](#) – or possibly even from a different database. Use the `executemany()` method again.

For this exercise add the `employees.csv` file, from the course [repository](#), to your "sql" directory (or the directory that contains your script).

```
# import from CSV

# import the csv library
import csv

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # open the csv file and assign it to a variable
    employees = csv.reader(open("employees.csv", "rU"))

    # create a new table called employees
    c.execute("CREATE TABLE employees(firstname TEXT, lastname TEXT)")

    # insert data into table
    c.executemany("INSERT INTO employees(firstname, lastname) values (?, ?)", employees)
```

Run the file. Now if you look in SQLite, you should see a new table called "employees" with 20 rows of data in it.

Try/Except

Remember this statement from above? "If you did not receive an error, then you can assume the code ran correctly." Well, what happens if you did see an error? We want to handle it gracefully. Let's refactor the code using Try/Except:

```
# INSERT Command with Error Handler

# import the sqlite3 library
import sqlite3

# create the connection object
conn = sqlite3.connect("new.db")

# get a cursor object used to execute SQL commands
cursor = conn.cursor()

try:
    # insert data
    cursor.execute("INSERT INTO populations VALUES('New York City', 'NY', 8400000)")
)
    cursor.execute("INSERT INTO populations VALUES('San Francisco', 'CA', 8000000)")
)

    # commit the changes
    conn.commit()
except sqlite3.OperationalError:
    print("Oops! Something went wrong. Try again...")

# close the database connection
conn.close()
```

Notice how we intentionally named the table "populations" instead of "population". Oops. Any idea how you could throw an exception, but also provide the user with relevant information about how to correct the issue? Google it!

Searching

Let's now look at how to retrieve data:

```
# SELECT statement

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # use a for loop to iterate through the database, printing the results line by
    line
    for row in c.execute("SELECT firstname, lastname from employees"):
        print(row)
```

Notice the `u` character in the output. This just stands for a Unicode string. [Unicode](#) is an international character encoding standard for displaying characters. This outputted because we printed the entire string rather than just the values.

Let's look at how to remove the unicode characters so we can output just the values:

```
# SELECT statement, remove unicode characters

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    c.execute("SELECT firstname, lastname from employees")

    # fetchall() retrieves all records from the query
    rows = c.fetchall()

    # output the rows to the screen, row by row
    for r in rows:
        print(r[0], r[1])
```

1. First, the `fetchall()` method retrieved all records from the query and stored them as a list of tuples.
2. We then assigned the records to the "rows" variable.

3. Finally, we printed the values using index notation, `print(r[0], r[1])` .

Updating and Deleting

This lesson covers how to use the UPDATE and DELETE SQL commands to change or delete records that match a specified criteria.

```
# UPDATE and DELETE statements

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # update data
    c.execute("UPDATE population SET population = 9000000 WHERE city='New York City'")

    # delete data
    c.execute("DELETE FROM population WHERE city='Boston'")

    print("\nNEW DATA:\n")

    c.execute("SELECT * FROM population")

    rows = c.fetchall()

    for r in rows:
        print(r[0], r[1], r[2])
```

1. We used the UPDATE command to change a specific field from a record and the DELETE command to delete an entire record.
2. We then displayed the results using the SELECT command.
3. We also introduced the WHERE clause, which is used to filter the results by a certain characteristic. You can also use this clause with the SELECT statement.

For example:

```
SELECT city from population WHERE state = 'CA'
```

This statement searches the database for cities where the state is CA. All other states are excluded from the query.

Homework

We covered a lot of material in the past few lessons. Be sure to go over it as many times as necessary before moving on.

Use three different scripts for these homework assignments:

- Using the "inventory" table from the previous homework assignment, add (`INSERT`) 5 records (rows of data) to the table. Make sure 3 of the vehicles are Fords while the other 2 are Hondas. Use any model and quantity for each.
 - [solution](#)
- Update the quantity on two of the records, and then output all of the records from the table.
 - [solution](#)
- Finally output only records that are for Ford vehicles.
 - [solution](#)

NOTE: If you have trouble the first time through with these assignments, don't go straight to the solution. Give yourself time to think and try again...and again...and again. This is the path to learning how to code.

Working with Multiple Tables

Now that you understand the basic SQL statements - SELECT, UPDATE, INSERT, and DELETE - let's add another layer of complexity by working with multiple tables. Before we begin, though, we need to add more records to the population table, as well as add one more table to the database.

```
# executemany() method

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # insert multiple records using a tuple
    # (you can copy and paste the values)
    cities = [
        ('Boston', 'MA', 600000),
        ('Los Angeles', 'CA', 38000000),
        ('Houston', 'TX', 2100000),
        ('Philadelphia', 'PA', 1500000),
        ('San Antonio', 'TX', 1400000),
        ('San Diego', 'CA', 130000),
        ('Dallas', 'TX', 1200000),
        ('San Jose', 'CA', 900000),
        ('Jacksonville', 'FL', 800000),
        ('Indianapolis', 'IN', 800000),
        ('Austin', 'TX', 800000),
        ('Detroit', 'MI', 700000)
    ]

    c.executemany("INSERT INTO population VALUES(?, ?, ?)", cities)

    c.execute("SELECT * FROM population WHERE population > 1000000")

    rows = c.fetchall()

    for r in rows:
        print(r[0], r[1], r[2])
```

Check SQLite to ensure the data was entered properly.

Did you notice the WHERE clause again? In this example, we chose to limit the results by only outputting cities with populations greater than one million.

Next, let's create a new table to use:

```
# Create a table and populate it with data

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    c.execute("""CREATE TABLE regions
                (city TEXT, region TEXT)
                """)

    # (again, copy and paste the values if you'd like)
    cities = [
        ('New York City', 'Northeast'),
        ('San Francisco', 'West'),
        ('Chicago', 'Midwest'),
        ('Houston', 'South'),
        ('Phoenix', 'West'),
        ('Boston', 'Northeast'),
        ('Los Angeles', 'West'),
        ('Houston', 'South'),
        ('Philadelphia', 'Northeast'),
        ('San Antonio', 'South'),
        ('San Diego', 'West'),
        ('Dallas', 'South'),
        ('San Jose', 'West'),
        ('Jacksonville', 'South'),
        ('Indianapolis', 'Midwest'),
        ('Austin', 'South'),
        ('Detroit', 'Midwest')
    ]

    c.executemany("INSERT INTO regions VALUES(?, ? )", cities)

    c.execute("SELECT * FROM regions ORDER BY region ASC")

    rows = c.fetchall()

    for r in rows:
        print(r[0], r[1])
```

We created a new table called "regions" that displayed the same cities with their respective regions. Notice how we used the ORDER BY clause in the SELECT statement to display the data in ascending order by region.

Open up the SQLite Browser to double check that the new table was in fact created and populated with data.

SQL Joins

The real power of relational tables comes from the ability to link data from two or more tables. This is achieved by using the JOIN command.

Let's write some code that will use data from both the "population" and the "regions" tables.

Code:

```
# JOINing data from multiple tables

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # retrieve data
    c.execute("""SELECT population.city, population.population,
                regions.region FROM population, regions
                WHERE population.city = regions.city""")

    rows = c.fetchall()

    for r in rows:
        print(r[0], r[1], r[2])
```

Take a look at the SELECT statement.

1. Since we are using two tables, fields in the SELECT statement must adhere to the following format: `table_name.column_name` (i.e., `population.city`).
2. In addition, to eliminate duplicates, as both tables include the city name, we used the WHERE clause as seen above.

Finally, let's organize the outputted results and clean up the code so it's more compact:

```
# JOINing data from multiple tables - cleanup

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    c.execute("""SELECT DISTINCT population.city, population.population,
                   regions.region FROM population, regions WHERE
                   population.city = regions.city ORDER by population.city ASC""")

    rows = c.fetchall()

    for r in rows:
        print("City: " + r[0])
        print("Population: " + str(r[1]))
        print("Region: " + r[2])
        print("")
```

Homework

- Add another table to accompany your "inventory" table called "orders". This table should have the following fields: "make", "model", and "order_date". Make sure to only include makes and models for the cars found in the inventory table. Add 15 records (3 for each car), each with a separate order date (YYYY-MM-DD).
 - [solution](#)
- Finally output the car's make and model on one line, the quantity on another line, and then the order_dates on subsequent lines below that.
 - [solution](#)

SQL Functions

SQLite has many built-in functions for aggregating and calculating data returned from a SELECT statement.

In this lesson, we will be working with the following functions:

Function	Result
AVG()	Returns the average value from a group
COUNT()	Returns the number of rows from a group
MAX()	Returns the largest value from a group
MIN()	Returns the smallest value from a group
SUM()	Returns the sum of a group of values

```
# SQLite Functions

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # create a dictionary of sql queries
    sql = {'average': "SELECT avg(population) FROM population",
            'maximum': "SELECT max(population) FROM population",
            'minimum': "SELECT min(population) FROM population",
            'sum': "SELECT sum(population) FROM population",
            'count': "SELECT count(city) FROM population"}

    # run each sql query item in the dictionary
    for keys, values in sql.items():

        # run sql
        c.execute(values)

        # fetchone() retrieves one record from the query
        result = c.fetchone()

        # output the result to screen
        print(keys + ":", result[0])
```

1. Essentially, we created a dictionary of SQL statements and then looped through the

- dictionary, executing each statement.
2. Next, using a for loop, we printed the results of each SQL query.

Homework

- Using the `COUNT()` function, calculate the total number of orders for each make and model.
 - [solution](#)
- Output the car's make and model on one line, the quantity on another line, and then the order count on the next line. The latter is a bit difficult, but please try it first before looking at my answer. **Remember: Google-it-first!**
 - [solution](#)

Example Application

We're going to end our discussion of the basic SQL commands by looking at an extended example. Please try the assignment first before reading the solution. The hardest part will be breaking it down into small, manageable bites. You've already learned the material; we're just putting it all together. Spend some time drawing out the workflow as a first step before writing any code.

In this application we will be performing aggregations on 100 integers.

Criteria:

1. Add 100 random integers, ranging from 0 to 100, to a new database called *newnum.db*.
2. Prompt the user whether they would like to perform an aggregation (AVG, MAX, MIN, or SUM) or exit the program altogether.

Break this assignment into two scripts. Name them *sql-insert.py* and *sql-search.py*.

Now stop for a minute and think about how you would set this up. Take out a piece of paper and *actually* write it out. Create a box for the first script and another box for the second. Write the criteria at the top of the page, and then begin by writing out *exactly* what the program should do in plain English in each box. These sentences will become the comments for your program.

First Script

Import libraries (we need the `random` library because of the random variable piece):

```
import sqlite3
import random
```

Establish a connection and create the *newnum.db* database:

```
with sqlite3.connect("newnum.db") as connection:
```

Open the cursor:

```
c = connection.cursor()
```

Create table called "numbers" with value "num" as an integer (the DROP TABLE command will remove the entire table if it exists so we can create a new one):

```
c.execute("DROP TABLE if exists numbers")
c.execute("CREATE TABLE numbers(num int)")
```

Use a `for` loop and the `random.randint()` method to insert 100 random values from 0 to 100:

```
for i in range(100):
    c.execute("INSERT INTO numbers VALUES(?)", (random.randint(0,100),))
```

Full Code:

```
# Assignment 3a - insert random data

# import the sqlite3 library
import sqlite3
import random

with sqlite3.connect("newnum.db") as connection:
    c = connection.cursor()

    # delete database table if exist
    c.execute("DROP TABLE if exists numbers")

    # create database table
    c.execute("CREATE TABLE numbers(num int)")

    # insert each number to the database
    for i in range(100):
        c.execute("INSERT INTO numbers VALUES(?)", (random.randint(0,100),))
```

Second Script

Again, start with writing out the steps in plain English.

1. Import the `sqlite3` library.
2. Connect to the database.
3. Establish a cursor.
4. Using an infinite loop, continue to ask the user to enter the number of an operation that they'd like to perform. If they enter a number associated with a SQL function,

run that function. However, if they enter number not associated with a function, ask them to enter another number. If they enter the number 5, break the loop and exit the program.

Clearly, step 4 needs to be broken up into multiple steps. Do that before you start writing any code.

Good luck!

Code:

Example Application

```
# Assignment 3b - prompt the user

# import the sqlite3 library
import sqlite3

# create the connection object
conn = sqlite3.connect("newnum.db")

# create a cursor object used to execute SQL commands
cursor = conn.cursor()

prompt = """
Select the operation that you want to perform [1-5]:
1. Average
2. Max
3. Min
4. Sum
5. Exit
"""

# loop until user enters a valid operation number [1-5]
while True:
    # get user input
    x = input(prompt)

    # if user enters any choice from 1-4
    if x in set(["1", "2", "3", "4"]):
        # parse the corresponding operation text
        operation = {1: "avg", 2:"max", 3:"min", 4:"sum"}[int(x)]

        # retrieve data
        cursor.execute("SELECT {}(num) from numbers".format(operation))

        # fetchone() retrieves one record from the query
        get = cursor.fetchone()

        # output result to screen
        print(operation + ":  %f" % get[0])

    # if user enters 5
    elif x == "5":
        print("Exit")

        # exit loop
        break
```

We asked the user to enter the operation they would like to perform (numbers 1 to 4), which queried the database and displayed either the average, minimum, maximum or sum (depending on the operation chosen). The loop continues forever until the user chooses 5 to break the loop.

SQL Summary

Basic SQL syntax...

Insert

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

Update

```
UPDATE table_name
SET column1=value1
WHERE some_column=some_value;
```

Delete

```
DELETE FROM table_name
WHERE some_column=some_value;
```

Chapter Summary

This chapter provided a brief summary of SQLite and how to use Python to interact with relational databases. There's a lot more you can do with databases that isn't covered here. If you'd like to explore relational databases further, there are a number of great resources online, like [ZetCode](#) and [tutorialspoint](#)'s Python MySQL Database Access.

Also, did you remember to commit and push to Github after each lesson?

Flask Blog App

Let's build a blog!

Requirements:

1. After a user logs in they are presented with all of the blog posts.
2. Users can add new, text-only blog entries from the same screen, read the entries themselves, or log out.

That's it.

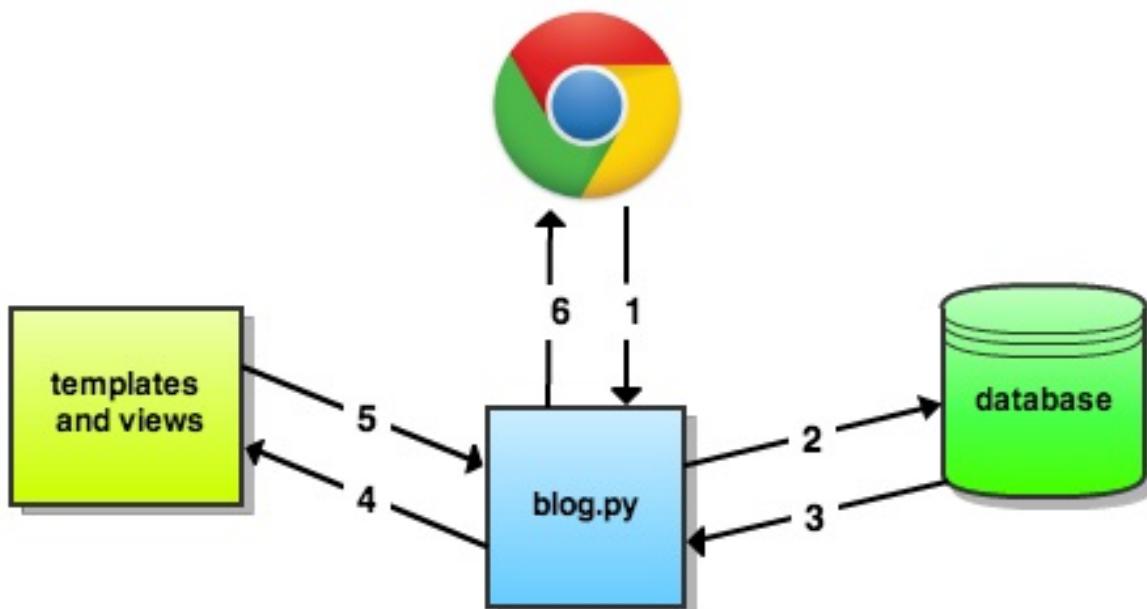
Project Structure

1. Within your "realpython" directory create a "flask-blog" directory.
2. Create and activate a virtual environment.
3. Make sure to place your app under version control by adding a Git repo.
4. Set up the following files and directories:

```
└── blog.py
└── static
    ├── css
    ├── img
    └── js
└── templates
```

First, all of the logic (Python/Flask code) goes in the *blog.py* file. The "static" directory holds static files (anything that is not dynamic) like JavaScript files, CSS stylesheets, and images. Finally, the "template" folder houses all of our HTML files.

The important thing to note is that the *blog.py* acts as the application controller. Flask works with a client-server model. The server receives HTTP requests from the client (the web browser), then returns content back to the client in the form of a response:



NOTE: HTTP is the method used for most web-based communications; the 'http://' (or 'https://') that prefixes URLs designates an HTTP request. Literally everything you see in your browser is transferred to you via HTTP.

5. Install flask:

```
$ pip install flask==0.11.1
```

Let's build our app!

Model

Our database has one table called *posts* with two fields - *title* and *post*. We can use the following script to create and populate the database:

```
# sql.py - Create a SQLite3 table and populate it with data

import sqlite3

# create a new database if the database doesn't already exist
with sqlite3.connect("blog.db") as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # create the table
    c.execute("""CREATE TABLE posts
                (title TEXT, post TEXT)
                """)

    # insert dummy data into the table
    c.execute('INSERT INTO posts VALUES("Good", "I\'m good.")')
    c.execute('INSERT INTO posts VALUES("Well", "I\'m well.")')
    c.execute('INSERT INTO posts VALUES("Excellent", "I\'m excellent.")')
    c.execute('INSERT INTO posts VALUES("Okay", "I\'m okay.")')
```

Save the file within your "flask-blog" directory as *sql.py*. Run it. This reads the database definition provided in *sql.py* and then creates the actual database schema and adds a number of entries. Check the SQLite Browser to ensure the table was created correctly and populated with data. *Notice how we escaped the apostrophes in the INSERT statements.*

Controller

Like the controller in the Quick Start app (`app.py`), this script will define the imports, configurations, and each view.

```
# blog.py - controller

# imports
from flask import Flask, render_template, request, session, \
    flash, redirect, url_for, g
import sqlite3

# configuration
DATABASE = 'blog.db'

app = Flask(__name__)

# pulls in app configuration by looking for UPPERCASE variables
app.config.from_object(__name__)

# function used for connecting to the database
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

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

Save this file as `blog.py` in your main project directory.

Make sure you understand how this is working:

```
# configuration
DATABASE = 'blog.db'

...snip...

# pulls in app configuration by looking for UPPERCASE variables
app.config.from_object(__name__)
```

The `from_object()` method takes an object as a parameter and passes it to `config`. In our case, because we passed `__name__`, it is passing in the entire module, which is `blog.py`. Behind the scenes, Flask looks for variables within the object passed to `config`.

that are defined using ALL CAPITAL LETTERS. So by defining `DATABASE` in `blog.py` we know that the Flask will find it and add it to our configuration for this app.

More information can be found in the [configuration](#) section, which is used for defining application-specific settings.

NOTE: Stop. You aren't cheating and using copy and paste - are you?

Views

After a user logs in, they are redirected to the main blog homepage where all posts are displayed. Users can also add posts from this page. For now, let's get the page set up, and worry about the functionality later.

```
{% extends "template.html" %}  
{% block content %}  
  <h2>Welcome to the Flask Blog!</h2>  
{% endblock %}
```

We also need a login page:

```
{% extends "template.html" %}  
{% block content %}  
  <h2>Welcome to the Flask Blog!</h2>  
  <h3>Please log in to access your blog.</h3>  
  <p>Temp Log in: <a href="/main">Log in</a></p>  
{% endblock %}
```

Save these files as *main.html* and *login.html*, respectively, in the "templates" directory. I know you have questions about the strange syntax - i.e., `{% extends "template.html" %}` - in both these files. We'll get to that in just a second.

Now update *blog.py* by adding two new functions for the views:

```
@app.route('/')  
def login():  
    return render_template('login.html')  
  
@app.route('/main')  
def main():  
    return render_template('main.html')
```

Updated code:

```
# blog.py - controller

# imports
from flask import Flask, render_template, request, session, \
    flash, redirect, url_for, g
import sqlite3

# configuration
DATABASE = 'blog.db'

app = Flask(__name__)

# pulls in configurations by looking for UPPERCASE variables
app.config.from_object(__name__)

# function used for connecting to the database
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

@app.route('/')
def login():
    return render_template('login.html')

@app.route('/main')
def main():
    return render_template('main.html')

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

In the first function, `login()`, we mapped the URL `/` to the function, which in turn sets the route to `login.html` in the templates directory. How about the `main()` function? What's going on there? Explain it to yourself by saying it out loud.

Templates

Templates are HTML skeletons that serve as the base for either your entire app or pieces of your app. They eliminate the need to code the basic HTML structure more than once. Separating templates from the main business logic (*blog.py*) helps with the overall organization.

Further, templates make it much easier to combine HTML and Python in a programmatic-like manner.

Let's start with a basic template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome, friends!</title>
  </head>
  <body>
    <div class="container">
      {% block content %}
      {% endblock %}
    </div>
  </body>
</html>
```

Save this as *template.html* within your templates directory.

There's a relationship between the parent, *template.html*, and child templates, *login.html* and *main.html*. This relationship is called **template inheritance**. Essentially, the child templates extend, or are a child of, *template.html*. This is achieved by using the `{% extends "template.html" %}` tag. This tag establishes the relationship between the template and views.

So, when Flask renders *main.html* it must first render *template.html*. And then renders *main.html* inside of it.

Notice that both the parent and child template files have identical block tags: `{% block content %}` and `{% endblock %}`. These define where the child templates, *login.html* and *main.html*, are filled in on the parent template. When Flask renders the parent template, *template.html*, the block tags are filled in with the code from the child templates:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome, friends!</title>
  </head>
  <body>
    <div class="container">
      {% block content %}
      {% endblock %}
    </div>
  </body>
</html>
```

```
{% extends "template.html" %}
{% block content %}
<div class="jumbo">
  <h2>Welcome to the Flask Blog!</h2>
  <h3>Please login to access your blog.</h3>
  <p>Temp Login: <a href="/main">Login</a></p>
</div>
{% endblock %}
```



Code found between the `{% %}` tags in the templates is a subset of Python used for basic expressions, like `for` loops or conditional statements. Meanwhile, variables, or the results from expressions, are surrounded by `{{ }}` tags.

SEE ALSO: There are a number of different [templating](#) formats. Flask by default uses [Jinja2](#) as its templating engine. Read more about Jinja2 templating from [this](#) blog post.

Sanity Check

Ready? Fire up your server (`python blog.py`), navigate to <http://localhost:5000/>, and let's run a test to make sure everything is working up to this point.

You should see the login page, and then if you click the link, you should be directed to the main page. If not, kill the server. Make sure all your files are saved. Try again. If you are still having problems, double-check your code against the above code snippets.

What's going on?

With `render_template()`, Flask immediately recognizes that `login.html` extends `template.html`. Flask renders `template.html`, then fills in the block tags, `{% block content %}` and `{% endblock %}`, with the HTML found in `login.html`.

User Login

Now that we have the basic structure set up, let's have some fun and add the blog's main functionality. Starting with the login page, set up a basic HTML form for users to log in to so that they can access the main blog page.

Add the following username and password variables to the configuration section of *blog.py*:

```
USERNAME = 'admin'  
PASSWORD = 'admin'
```

Also in the configuration section, add the `SECRET_KEY`, which is used for managing user sessions:

```
SECRET_KEY = 'hard_to_guess'
```

WARNING: Make the value of your secret key really, really hard, if not impossible, to guess. Use a random key generator to do this. Never, ever use a value you pick on your own. Or you can use your OS to get a random string:

```
>>> import os  
>>> os.urandom(24)  
'rM\xb1\xdc\x12o\xd6i\xff+9$T\x8e\xec\x00\x13\x82.*\x16TG\xbd'
```

Now you can simply assign that string to the secret key: `SECRET_KEY = rM\xb1\xdc\x12o\xd6i\xff+9$T\x8e\xec\x00\x13\x82.*\x16TG\xbd`

Updated *blog.py* configuration:

```
# configuration  
DATABASE = 'blog.db'  
USERNAME = 'admin'  
PASSWORD = 'admin'  
SECRET_KEY = 'hard_to_guess'
```

Update the `login()` function in the *blog.py* file to match the following code:

```
@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    status_code = 200
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME'] or \
           request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid Credentials. Please try again.'
            status_code = 401
    else:
        session['logged_in'] = True
    return redirect(url_for('main'))
return render_template('login.html', error=error), status_code
```

This function compares the username and password entered against those from the configuration section. If the correct username and password are entered, the user is redirected to the main page and the session key, `logged_in`, is set to `True`. If the wrong information is entered, an error message is flashed to the user.

NOTE: Notice how we had to specify a `POST` request. By default, routes are set up automatically to handle GET requests. If you need to add different HTTP methods, such as a POST, you must add the methods argument to the decorator.

The `url_for()` function generates an endpoint for the provided method.

For example:

```
>>> from flask import Flask, url_for
>>> app = Flask(__name__)
>>> @app.route('/')
... def index():
...     pass
...
>>> @app.route('/main')
... def main():
...     pass
...
>>> with app.test_request_context():
...     print(url_for('index'))
...     print(url_for('main'))
...
/
/main
```

NOTE: The `test_request_context()` method is used to mock an actual request for testing purposes. Used in conjunction with the `with` statement, you can activate the request to temporarily access session objects to test. More on this later.

Looking back at the code from above, walk through this function, `login()`, line by line, saying *out loud* what each line accomplishes.

Now, we need to update `login.html` to include the HTML form:

```
{% extends "template.html" %}  
{% block content %}  
  <div class="login-wrapper">  
    <h2>Welcome to the Flask Blog!</h2>  
    <h3>Please log in to access your blog.</h3>  
    <form action="" method="post">  
      <div>  
        <label for="name">Username:</label>  
        <input type="text" id="username" name="username" value="{{  
          request.form.username }}"/>  
      </div>  
      <div>  
        <label for="mail">Password:</label>  
        <input type="password" id="password" name="password" value="{{  
          request.form.password }}"/>  
      </div>  
      <div>  
        <button type="submit">Log In</button>  
      </div>  
    </form>  
  </div>  
{% endblock %}
```

If you are unfamiliar with how HTML forms operate, please visit this [link](#).

Next, add a function for logging out to `blog.py`:

```
@app.route('/logout')  
def logout():  
    session.pop('logged_in', None)  
    flash('You were logged out')  
    return redirect(url_for('login'))
```

The `logout()` function uses the `pop()` function to reset the session key to the default value when the user logs out. The user is then redirected back to the login screen and a message is flashed indicating that they were logged out.

NOTE: The `pop()` function used here is defined within the `session` class. It is not the `pop()` function native to python that is used on *lists*.

Add the following code to the `template.html` file, just before the content tag, `{% block content %}` -

```
{% for message in get_flashed_messages() %}  
  <div class="flash">{{ message }}</div>  
{% endfor %}  
{% if error %}  
  <p class="error"><strong>Error:</strong> {{ error }}</p>  
{% endif %}
```

-so that the template now looks like this:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Welcome, friends!</title>  
  </head>  
  <body>  
    <div class="container">  
      {% for message in get_flashed_messages() %}  
        <div class="flash">{{ message }}</div>  
      {% endfor %}  
      {% if error %}  
        <p class="error"><strong>Error:</strong> {{ error }}</p>  
      {% endif %}  
      <!-- inheritance -->  
      {% block content %}  
      {% endblock %}  
      <!-- end inheritance -->  
    </div>  
  </body>  
</html>
```

Finally, add a log out link to the `main.html` page:

```
{% extends "template.html" %}  
{% block content %}  
  <h2>Welcome to the Flask Blog!</h2>  
  <p><a href="{{ url_for('logout') }}">Log out</a></p>  
{% endblock %}
```

View it! Fire up the server. Manually test everything out. Make sure you can log in and log out and that the appropriate messages are displayed, depending on the situation.

Sessions

Now that users are able to log in and log out, we need to protect *main.html* from unauthorized access. Currently, it can be accessed without logging in. Go ahead and see for yourself: Launch the server and point your browser at <http://localhost:5000/main>. See what I mean?

To prevent unauthorized access to *main.html*, we need to set up sessions, as well as utilize the `login_required` decorator.

[Sessions](#) store user information in a secure manner, usually as a token, within a client-side cookie. In this case, when the session key, `logged_in`, is set to `True`, the user has the rights to view the *main.html* page. Go back and take a look at the `login()` function so you can see this logic.

The `login_required` decorator, meanwhile, checks to make sure that a user is authorized before allowing access to certain pages. To implement this, we will set up a new function which will be used to restrict access to *main.html*.

Start by importing `functools` within your controller, *blog.py*:

```
from functools import wraps
```

[Functools](#) is a module used for extending the capabilities of functions with other functions, which is exactly what decorators accomplish.

First, set up the new function in *blog.py*:

```
def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return test(*args, **kwargs)
        else:
            flash('You need to log in first.')
            return redirect(url_for('login'))
    return wrap
```

This checks to see if `logged_in` is in the session. If it is, then we call the appropriate function (e.g., the function that the decorator is applied to), and if not, the user is redirected back to the login screen with a message stating that a log in is required.

Add the decorator to the top of the `main()` function:

```
@app.route('/main')
@login_required
def main():
    return render_template('main.html')
```

Updated code:

```
# blog.py - controller

# imports
from flask import Flask, render_template, request, session, \
    flash, redirect, url_for, g
import sqlite3
from functools import wraps

# configuration
DATABASE = 'blog.db'
USERNAME = 'admin'
PASSWORD = 'admin'
SECRET_KEY = 'hard_to_guess'

app = Flask(__name__)

# pulls in configurations by looking for UPPERCASE variables
app.config.from_object(__name__)

# function used for connecting to the database
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return test(*args, **kwargs)
        else:
            flash('You need to log in first.')
            return redirect(url_for('login'))
    return wrap

@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    status_code = 200
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME'] or \
```

```

        request.form['password'] != app.config['PASSWORD']:
    error = 'Invalid Credentials. Please try again.'
    status_code = 401
else:
    session['logged_in'] = True
    return redirect(url_for('main'))
return render_template('login.html', error=error), status_code

@app.route('/main')
@login_required
def main():
    return render_template('main.html')

@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('login'))

if __name__ == '__main__':
    app.run(debug=True)

```

When a GET request is sent to access *main.html* to view the HTML, it first hits the `@login_required` decorator and the entire function, `main()`, is momentarily replaced (or wrapped) by the `login_required()` function. Then when the user is logged in, the `main()` function is invoked, allowing the user to access *main.html*. If the user is not logged in, they are redirected back to the login screen.

Test this out. But first, did you notice in the terminal that you can see the client requests as well as the server responses? After you perform each test, check the server responses:

1. Log in successful:

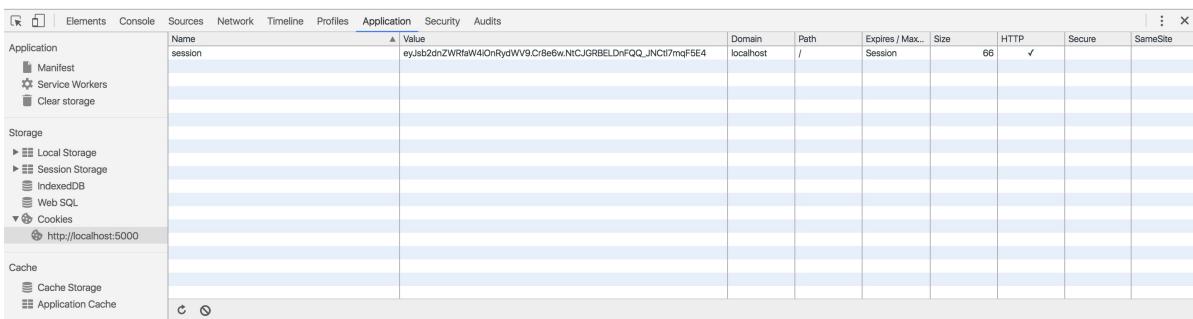
```

127.0.0.1 - - [17/Sep/2016 12:37:33] "POST / HTTP/1.1" 302 -
127.0.0.1 - - [17/Sep/2016 12:37:33] "GET /main HTTP/1.1" 200 -

```

Here, the credentials were sent with the POST request, and the server responded with a 302, redirecting the user to *main.html*. The GET request sent to access *main.html* was successful, as the server responded with a 200.

Once logged in, the session token is stored in a client-side cookie. You can view this token in your browser by opening up Developer Tools in Chrome, clicking the "Application" tab, then in the side bar underneath the "Storage" heading click on cookies. You should see this:



Application	Name	Value	Domain	Path	Expires / Max...	Size	HTTP	Secure	SameSite
	session	eyJsb2dnZWltZW4lOnRydWV9.Cr8e6w.NtCJGRBELDnFQQ_JNCt7mqF5E4	localhost	/	Session	66	✓		

2. Log out:

```
127.0.0.1 - - [17/Sep/2016 12:37:33] "GET /logout HTTP/1.1" 302 -
127.0.0.1 - - [17/Sep/2016 12:37:33] "GET / HTTP/1.1" 200 -
```

When you log out, a GET request is sent and the subsequent response redirects you back to *login.html*. Again, this request was successful.

3. Log in failed:

```
127.0.0.1 - - [17/Sep/2016 12:37:35] "POST / HTTP/1.1" 401 -
```

If you enter the wrong credentials you get a *401* ('not authorized') because that is what we told the login method to return on a failed login attempt. You might want to go back up and look at the *login route* to re-examine how this is working.

4. Attempt to access <http://localhost:5000/main> without first logging in:

```
127.0.0.1 - - [17/Sep/2016 12:37:37] "GET /main HTTP/1.1" 302 -
127.0.0.1 - - [17/Sep/2016 12:37:37] "GET / HTTP/1.1" 200 -
```

Here we got a *302* which means *redirect* and a *200* because we hit the login page successfully.

The server stack trace comes in handy when you need to debug your code. Let's say you forgot to add the redirect to the `login()` function, `return redirect(url_for('main'))`. If you glance at your code and can't figure out what's going on, the server log may provide a hint:

```
127.0.0.1 - - [17/Sep/2016 12:37:44] "POST / HTTP/1.1" 200 -
```

You can see that the POST request was successful, but nothing happened after. This should give you enough of a hint to know what to do. This is a rather simple case, but you will find that when your codebase grows just how handy the server log can be when debugging.

Show Posts

Now that basic security is set up, we need to display some information to the user. Start by displaying the current posts.

Update the `main()` function within `blog.py`:

```
@app.route('/main')
@login_required
def main():
    g.db = connect_db()
    cur = g.db.execute('select * from posts')
    posts = [dict(title=row[0], post=row[1]) for row in cur.fetchall()]
    g.db.close()
    return render_template('main.html', posts=posts)
```

What's going on?

1. `g.db = connect_db()` connects to the database.
2. `cur = g.db.execute('select * from posts')` then fetches data from the `posts` table within the database.
3. `posts = [dict(title=row[0], post=row[1]) for row in cur.fetchall()]` creates an array of dictionaries. Each dictionary contains the data from a row retrieved from the `posts` table. This array is assigned to the variable `posts`.
4. `posts=posts` passes that variable to the `main.html` file.

NOTE: We are using the `g` object as a 'request blackboard'. This blackboard gets wiped clean after the request is finished. Don't work about this concept, just think of it as a best practice to use the `g` inside of requests when dealing with database connections. For more information have a look at this [Stack Overflow post](#).

Edit `main.html` to loop through the dictionary in order to display the titles and posts:

```
{% extends "template.html" %}  
{% block content %}  
  <h2>Welcome to the Flask Blog!</h2>  
  <p><a href="{{ url_for('logout') }}>Log out</a></p>  
  <br/>  
  <br/>  
  <h3>Posts:</h3>  
  {% for p in posts %}  
    <strong>Title:</strong> {{ p.title }} <br/>  
    <strong>Post:</strong> {{ p.post }} <br/>  
    <br/>  
  {% endfor %}  
{% endblock %}
```

This is a relatively straightforward example: We passed in the `posts` variable from `blog.py` that contains the data fetched from the database. Then, we used a simple `for` loop to iterate through the list to display the results.

Check this out in our browser!

Add Posts

Finally, users need the ability to add new posts. We can start by adding a new function to `blog.py` called `add()`:

```
@app.route('/add', methods=['POST'])
@login_required
def add():
    title = request.form['title']
    post = request.form['post']
    if not title or not post:
        flash("All fields are required. Please try again.")
        return redirect(url_for('main'))
    else:
        g.db = connect_db()
        g.db.execute('insert into posts (title, post) values (?, ?)',
                    [request.form['title'], request.form['post']])
        g.db.commit()
        g.db.close()
        flash('New entry was successfully posted!')
        return redirect(url_for('main'))
```

First, we used an `if` statement to ensure that all fields are populated with data. Then, the data is added, as a new row, to the database table.

NOTE: The above description is a high-level overview of what's really happening.

Get granular with it. Read what each line accomplishes out loud.

Next, add the HTML form to the `main.html` page:

```
{% extends "template.html" %}  
{% block content %}  
    <h2>Welcome to the Flask Blog!</h2>  
    <p><a href="{{ url_for('logout') }}>Log out</a></p>  
    <h3>Add a new post:</h3>  
    <form action="{{ url_for('add') }}" method="post" class="add">  
        <div>  
            <label>Title:</label>  
            <input name="title" type="text">  
        </div>  
        <div>  
            <label>Post:</label>  
            <textarea name="post" rows="5" cols="40"></textarea>  
        </div>  
        <div class="">  
            <input class="button" type="submit" value="Save">  
        </div>  
    </form>  
    <br>  
    <br>  
    <h3>Posts:</h3>  
    {% for p in posts %}  
        <strong>Title:</strong> {{ p.title }} <br>  
        <strong>Post:</strong> {{ p.post }} <br>  
        <br>  
    {% endfor %}  
{% endblock %}
```

Test this out! Make sure to add a post.

We sent an HTTP POST request when the form was submitted, which was handled on the server-side by the `add()` view function. This, in turn, redirected us back to `main.html` with the new post:

```
127.0.0.1 - - [01/Feb/2014 12:14:24] "POST /add HTTP/1.1" 302 -  
127.0.0.1 - - [01/Feb/2014 12:14:24] "GET /main HTTP/1.1" 200 -
```

Style

Now that the app is working properly, let's make it look a bit nicer. To do this, we can edit the HTML and CSS. *We will be covering HTML and CSS in more depth in a later chapter. If you are unfamiliar with either of them, just follow along for now.*

```
.container {  
  background: #f4f4f4;  
  margin: 2em auto;  
  padding: 0.8em;  
  width: 30em;  
  border: 2px solid #000;  
}  
  
.login-wrapper {  
  text-align: center;  
}  
  
label {  
  display: block;  
  margin-bottom: 5px;  
  font-weight: bold;  
}  
  
.login-wrapper label {  
  display: inline-block;  
}  
  
input {  
  margin-bottom: 10px;  
}  
  
button {  
  padding: 5px;  
  margin: 10px 0;  
}  
  
.flash, .error {  
  background: #000;  
  color: #fff;  
  padding: 0.5em;  
}
```

Save this as `styles.css` and place it in your "static/css" directory. Then add a link to the external stylesheet within the head, `<head> </head>`, of the `template.html` file:

```
<link rel="stylesheet"
      href="{{ url_for('static', filename='css/styles.css') }}">
```

This tag is fairly straightforward. Essentially, the `url_for()` function generates a URL to the `styles.css` file, saying, "Look in the static folder for `styles.css`".



Feel free to play around with the CSS more if you'd like. If you do, send us the CSS, so we can make it look better.

Conclusion

Let's recap:

1. First, we used Flask to create a basic app structure.
2. Then we added a log in form.
3. After that, we added sessions and the `login_required` decorator to prevent unauthorized access to the *main.html* page.
4. Next, we fetched data from SQLite to show all the blog posts, then added the ability for users to add new posts.
5. Finally, we added some basic CSS styles.

Make sure to commit your code to Git and then push to Github!

Interlude: Debugging in Python

When solving complicated coding problems, it's important to use an interactive debugger for examining executed code line by line. Python provides such a tool called [pdb](#) (the "Python DeBugger") within the standard library, where you can set breakpoints, step through your code, and inspect the stack trace.

Always keep in mind that while `pdb`'s primary purpose is debugging code, it's more important that you *understand* what's happening in your code while debugging. This in itself will help with debugging.

Workflow

Let's look at a simple example.

Save the following code as `pdb_ex.py` in a new folder called "debugging":

```
import sys
from random import choice

random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

while True:
    print("To exit this game type 'exit'")
    answer = input("What is {} times {}? ".format(
        choice(random2), choice(random1)))

    # exit
    if answer == "exit":
        print("Now exiting game!")
        sys.exit()

    # determine if number is correct
    elif answer == choice(random2) * choice(random1):
        print("Correct!")
    else:
        print("Wrong!")
```

Run it. See the problem? There's either an issue with the multiplication or the logic within the `if` statement.

Let's debug!

Import the `pdb` module:

```
import pdb
```

Next, add `pdb.set_trace()` within the `while` loop to set your first breakpoint:

```
import sys
import pdb
from random import choice

random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

while True:
    print("To exit this game type 'exit'")

    pdb.set_trace()

    answer = input("What is {} times {}? ".format(
        choice(random2), choice(random1)))

    # exit
    if answer == "exit":
        print("Now exiting game!")
        sys.exit()

    # determine if number is correct
    elif answer == choice(random2) * choice(random1):
        print("Correct!")
    else:
        print("Wrong!")
```

When you run the code you should see the following output:

```
$ python pdb_ex.py
To exit this game type 'exit'
> /debugging/pdb_ex.py(13)<module>()
-> answer = input("What is {} times {}? ".format(
(Pdb)
```

Essentially, when the Python interpreter runs `pdb.set_trace()`, execution stops, and you'll see the next line in the program as well as a prompt waiting for input.

From here you step through your code to see what happens, line by line. You have access to a number of [commands](#), which can be daunting at first, but on a day-to-day basis you'll only use a few of them:

- `n` : step forward one line
- `p <variable name>` : prints the current value of the provided variable
- `l` : displays the entire program along with where the current break point is
- `q` : exits the debugger and the program
- `c` : exits the debugger and the program continues to run

- `b <line #>` : adds a breakpoint at a specific line #

NOTE If you don't remember the list of commands you can always type `?` or `help` to see the entire list.

Let's debug this together...

First, see what the value of `answer` is:

```
(Pdb) n
> /debugging/pdb_ex.py(14)<module>()
-> choice(random2), choice(random1))
(Pdb) n
What is 12 times 10? 120
> /debugging/pdb_ex.py(17)<module>()
-> if answer == "exit":
(Pdb) p answer
'120'
```

Next, let's continue through the program to see if that value, `120`, changes:

```
(Pdb) n
> /debugging/pdb_ex.py(22)<module>()
-> elif answer == choice(random2) * choice(random1):
(Pdb) n
> /debugging/pdb_ex.py(25)<module>()
-> print("Wrong!")
(Pdb) p answer
'120'
```

Since the answer does not change there must be something wrong with the program logic in the `if` statement, starting with the `elif`.

Update the code for testing:

```
import sys
from random import choice

random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

while True:
    print("To exit this game type 'exit'")
    answer = input("What is {} times {}? ".format(
        choice(random2), choice(random1)))
    )

    # exit
    if answer == "exit":
        print("Now exiting game!")
        sys.exit()

    test = int(choice(random2))*int(choice(random1))
    # determine if number is correct
    # elif answer == choice(random2) * choice(random1):
    #     print("Correct!")
    # else:
    #     print("Wrong!")
```

We assigned the value we are using to test our `answer` against to the variable `test`.

Debug time!

Exit the debugger, and let's start over.

```
$ python pdb_ex.py
To exit this game type 'exit'
> /debugging/pdb_ex.py(13)<module>()
-> answer = input("What is {} times {}? ".format(
(Pdb) n
> /debugging/pdb_ex.py(14)<module>()
-> choice(random2), choice(random1)))
(Pdb) n
What is 2 times 2? 4
> /debugging/pdb_ex.py(17)<module>()
-> if answer == "exit":
(Pdb) n
> /debugging/pdb_ex.py(21)<module>()
-> test = int(choice(random2))*int(choice(random1))
(Pdb) n
> /debugging/pdb_ex.py(8)<module>()
-> while True:
(Pdb) p test
20
```

There's our answer. The value in the `elif` varies from the `answer` value, `20` vs. `4`. Thus, the `elif` will always return "Wrong!".

Refactor:

```
import sys
from random import choice

random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

while True:
    print("To exit this game type 'exit'")
    num1 = choice(random2)
    num2 = choice(random1)
    answer = int(input("What is {} times {}? ".format(num1, num2)))

    # exit
    if answer == "exit":
        print("Now exiting game!")
        sys.exit()

    # determine if number is correct
    elif answer == num1 * num2:
        print("Correct!")
        break
    else:
        print("Wrong!")
```

Ultimately, the program was generating new numbers for comparison within the `elif`, causing the user input to be wrong each time. Additionally, in the comparison - `elif answer == num1 * num2` - the `answer` is a string while `num1` and `num2` are integers. To fix this, you just need to caste the answer to an integer.

Breakpoints

One thing we didn't touch on is setting breakpoints, which allow you to pause code execution at a certain line. To set a breakpoint while debugging, you simply call the `break` command and then add the line number that you wish to break on - `b <line #>`

Simple example:

```
import pdb

def add_one(num):
    result = num + 1
    print(result)
    return result

def main():
    pdb.set_trace()
    for num in range(0, 10):
        add_one(num)

if __name__ == "__main__":
    main()
```

Save this as `pdb_ex2.py`.

Now, let's look at an example:

```
python pdb_ex2.py
> /debugging/pdb_ex2.py(12)main()
-> for num in range(0, 10):
(Pdb) b 5
Breakpoint 1 at /debugging/pdb_ex2.py:5
(Pdb) c
> /debugging/pdb_ex2.py(5)add_one()
-> result = num + 1
(Pdb) args
num = 0
(Pdb) b 13
Breakpoint 2 at /debugging/pdb_ex2.py:13
(Pdb) c
1
> /debugging/pdb_ex2.py(13)main()
-> add_one(num)
(Pdb) b 5
Breakpoint 3 at /debugging/pdb_ex2.py:5
(Pdb) c
> /debugging/pdb_ex2.py(5)add_one()
-> result = num + 1
(Pdb) args
num = 1
(Pdb) c
2
> /debugging/pdb_ex2.py(13)main()
-> add_one(num)
```

Here, we started the debugger on line 11, then set a breakpoint on line 5. We continued the program until it hit that breakpoint. Then we checked the value of `num` , `0` . We set another break at line 13, then continued again and said that the result was `1 - result = 0 + 1` - which is what we expected. Then we did the same process again and found that the next result was 2 based on the value of `num - result = 1 + 1` .

Hope that makes sense.

Post Mortem Debugging

You can also use pdb to debug code that's already crashed, after the fact. Take the following code, for example:

```
def add_one_hundred():
    again = 'yes'
    while again == 'yes':
        number = input('Enter a number between 1 and 10: ')
        new_number = (int(number) + 100)
        print('{} plus 100 is {}'.format(number, new_number))
        again = input('Another round, my friend? ("yes" or "no") ')
    print("Goodbye!")
```

Save this as *post_mortem_pdb.py*.

This function simply adds 100 to a number inputted by the user, then outputs the results to the screen.

What happens if you enter a string instead of an integer?

```
>>> from post_mortem_pdb import *
>>> add_one_hundred()
Enter a number between 1 and 10: test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "post_mortem_pdb.py", line 5, in add_one_hundred
    new_number = (int(number) + 100)
ValueError: invalid literal for int() with base 10: 'test'
>>>
```

NOTE: Here, Python provided us with a **traceback**, which is really just the details about what happened when it encountered the error. Most of the time, the error messages associated with a traceback are very helpful.

Now we can use pdb to start debugging where the exception occurred:

```
>>> import pdb; pdb.pm()
> /debugging/post_mortem_pdb.py(5)add_one_hundred()
-> new_number = (int(number) + 100)
(Pdb)
```

Start debugging!

So we know that the line `new_number = (int(number) + 100)` broke the code - because you can't convert a string to an integer.

This is a simple example, but imagine how useful this would be in a large program with multiple scripts that you can't fully visualize. You can immediately jump back into the program where the exception was thrown and start the debugging process. This can be incredibly useful. We'll look at an example of just that when we start working with the advanced Flask and Django material later in the course.

Cheers!

Homework

- Watch [this](#) video and [this](#) video on debugging
- Read [An Introduction to Python Debugging](#)

Flask: FlaskTaskr, Part 1 - Quick Start

In this section, we'll develop a task management system aptly named *FlaskTaskr*. We'll start by creating a simple app following a similar workflow to the Flask blog app and then extend the app by adding a number of features and extensions in order to make this a full-featured web application.

Let's get to it.

To start, this application has the following features:

1. Users can sign in and out from the landing page
2. New users can register on a separate registration page
3. Once signed in, users can add new tasks (each task consists of a name, due date, priority, status, and a unique ID)
4. Users can view all incomplete tasks from the same page
5. Users can also delete tasks and mark tasks as complete (deleted tasks will be removed from the database)

Before beginning, take a moment to review the steps used to create the blog application. Again, we'll be using a similar process - but it will go much faster. *Make sure to commit your changes to your local Git repo and push to Github frequently.*

Getting Started

For simplicity's sake, since you should be familiar with the workflow, explanations will not address what has already been explained.

Navigate to your "realpython" directory, and create a new directory called "flasktaskr".

Navigate into the newly created directory. Create and activate a new virtual environment.

Install Flask:

```
$ pip install flask==0.11.1
$ pip freeze > requirements.txt
```

Add a new directory called "project", which is the project root directory. Create the following files and directories within the root directory:

```
└── static
    ├── css
    ├── img
    └── js
└── templates
└── views.py
```

Add a Git repo to the "flasktaskr" directory - `git init`

NOTE: In the first two Flask apps, we utilized a single-file structure. This is fine for small apps, but as your project scales, this structure will become much more difficult to maintain. It's a good idea to break your app into several files, each handling a different set of responsibilities to separate out concerns. The overall structure follows the Model-View-Controller architecture pattern.

Configuration

Remember how we placed all of the blog app's configuration directly in the main controller? Again, it's best practice to actually place these in a separate file, and then import that file into the controller. There's a number of reasons for this, but in the end, it separates our app's logic from configuration and static variables.

Create a configuration file called `_config.py` and save it in the project root:

```
import os

# grab the folder where this script lives
basedir = os.path.abspath(os.path.dirname(__file__))

DATABASE = 'flasktaskr.db'
USERNAME = 'admin'
PASSWORD = 'admin'
WTF_CSRF_ENABLED = True
SECRET_KEY = 'myprecious'

# define the full path for the database
DATABASE_PATH = os.path.join(basedir, DATABASE)
```

What's happening?

1. The `WTF_CSRF_ENABLED` config setting is used for [cross-site request forgery](#) prevention, which makes your app more secure. This setting is used by the [Flask-WTF](#) extension.
2. The `SECRET_KEY` config setting is used in conjunction with the `WTF_CSRF_ENABLED` setting in order to create a cryptographic token that is used to validate a form. It's also used for the same reason in conjunction with sessions. Make sure the secret key is nearly impossible to guess. Use a [random key generator](#).

Database

Based on the main functionality - each task consists of a name, due date, priority, status, and a unique ID - we need one database table, consisting of these fields - `task_id`, `name`, `due_date`, `priority`, and `status`. The value of `status` will either be a `1` or `0` - `1` if the task is open and `0` if closed.

```
# project/db_create.py

import sqlite3
from _config import DATABASE_PATH

with sqlite3.connect(DATABASE_PATH) as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # create the table
    c.execute("""CREATE TABLE tasks(task_id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL, due_date TEXT NOT NULL, priority INTEGER NOT NULL,
        status INTEGER NOT NULL)""")

    # insert dummy data into the table
    c.execute(
        'INSERT INTO tasks (name, due_date, priority, status)'
        'VALUES("Finish this tutorial", "03/25/2015", 10, 1)'
    )
    c.execute(
        'INSERT INTO tasks (name, due_date, priority, status)'
        'VALUES("Finish Real Python Course 2", "03/25/2015", 10, 1)'
    )
```

Two things to note:

1. Notice how we did not specify the `task_id` when adding rows into the table as it's an auto-incremented value. Also, we used a status of `1` to indicate that each of the tasks are considered "open" tasks. This is a default value.
2. We imported the `DATABASE_PATH` variable from the configuration file we created just a second ago.

Save this in the "project" directory as `db_create.py` and run it. Was the database created? Did it populate with data? How do you check? SQLite Browser. Test it out.

Controller

Add the following code to `views.py`:

```
# project/views.py

import sqlite3
from functools import wraps

from flask import Flask, flash, redirect, render_template, \
    request, session, url_for

# config

app = Flask(__name__)
app.config.from_object('_config')

# helper functions

def connect_db():
    return sqlite3.connect(app.config['DATABASE_PATH'])

def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return test(*args, **kwargs)
        else:
            flash('You need to login first.')
            return redirect(url_for('login'))
    return wrap

# route handlers

@app.route('/logout/')
def logout():
    session.pop('logged_in', None)
    flash('Goodbye!')
    return redirect(url_for('login'))

@app.route('/', methods=['GET', 'POST'])
def login():
```

```

if request.method == 'POST':
    if request.form['username'] != app.config['USERNAME'] \
        or request.form['password'] != app.config['PASSWORD']:
        error = 'Invalid Credentials. Please try again.'
        return render_template('login.html', error=error)
    else:
        session['logged_in'] = True
        flash('Welcome!')
        return redirect(url_for('tasks'))
return render_template('login.html')

```

Save the file.

What's happening?

You've seen this all before, but let's quickly review:

1. Right now, we have one view, `login`, which is mapped to the main URL, `/`.
2. Sessions are configured, adding a value of `True` to the `logged_in` key, which is removed (via the `pop` method) when the user logs out.
3. The `login_required()` decorator is also configured. Do you remember how that works? You can see that after a user logs in, they will be redirected to `tasks`, which still needs to be specified.

NOTE: Refer to the *Flask Blog App* chapter for further explanation on any details of this code that you do not understand.

Commit your code. Make sure to add a `.gitignore` file to the "flasktaskr" directory:

```

env
venv
*.pyc
__pycache__
*.DS_Store
project/_config.py

```

Let's set up the login and base templates as well as an external stylesheet.

Templates and Styles

Login template (child)

```
{% extends "_base.html" %}

{% block content %}

<h1>Welcome to FlaskTaskr.</h1>
<h3>Please login to access your task list.</h3>

<form action="/" method="post">
    <div class="input-group">
        <label for="name">Username:</label>
        <input type="text" id="username" name="username" value="{{ request.form.username }}">
    </div>
    <div class="input-group">
        <label for="mail">Password:</label>
        <input type="password" id="password" name="password" value="{{ request.form.password }}"/>
    </div>
    <div>
        <button type="submit">Login</button>
    </div>
</form>

<p><em>Use 'admin' for the username and password.</em></p>

{% endblock %}
```

Save this as *login.html* in the "templates" directory.

Base template (parent)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to FlaskTaskr!!</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/main.css') }}">

  </head>
  <body>
    <div class="page">

      {% for message in get_flashed_messages() %}
        <div class="flash">{{ message }}</div>
      {% endfor %}

      {% if error %}
        <div class="error"><strong>Error:</strong> {{ error }}</div>
      {% endif %}

      <br>

      {% block content %}
      {% endblock %}

    </div>
  </body>
</html>
```

Save this as `_base.html` in the "templates" directory. Remember the relationship between the parent and child templates discussed in the blog chapter? Read more about it [here](#), and check out [this](#) blog post for more information.

Stylesheet

For now we'll *temporarily* "borrow" the majority of the stylesheet from the Flask [tutorial](#). Find the `main.css` file in the `book2-exercises` in the `assets/flasktaskr-01` directory. Copy and paste this code into a new file, and then save it as `main.css` in the "css" directory within the "static" directory.

Sanity Check

Instead of running the application directly from the controller (like in the blog app), let's create a separate file. Why? In order to remove unnecessary code from the controller that does not pertain to the *actual* business logic. Again, we're separating out concerns.

```
# project/run.py

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

Save this as *run.py* in your "project" directory. The directory structure within the *project* directory should now look like this:

```
└── _config.py
└── db_create.py
└── flasktaskr.db
└── run.py
└── static
    ├── css
    │   └── main.css
    ├── img
    └── js
└── templates
    ├── _base.html
    └── login.html
└── views.py
```

Fire up the server:

```
$ python run.py
```

Navigate to <http://localhost:5000/>. Make sure everything works thus far. You'll only be able to view the styled login page (but not login) as we have not set up the *tasks.html* page yet. Essentially, we're ensuring that the-

- App runs
- Templates are working correctly
- Basic logic in *views.py* works

Welcome to FlaskTaskr.

Please login to access your task list.

Username:

Password:

Use 'admin' for the username and password.

So, we've split out our app into separate files, each with specific responsibilities:

1. `_config.py`: holds our app's settings and configuration global variables
2. `views.py`: contains the business logic - e.g., the routing rules - and sets up our Flask app (the latter of which could actually be moved to a different file)
3. `db_create.py`: sets up and creates the database
4. `run.py`: starts and runs the Flask dev server

Remember when all those responsibilities were crammed into one single file? This new structure should be much easier to follow.

Now is a good time to commit your code to Git.

Tasks

1. Once signed in, users can add new tasks
2. Users can view all incomplete tasks from the same page
3. Users can also delete tasks and mark tasks as complete

Add the following route handler and view function to the `views.py` file:

```
@app.route('/tasks/')
@login_required
def tasks():
    g.db = connect_db()
    cursor = g.db.execute(
        'select name, due_date, priority, task_id from tasks where status=1'
    )
    open_tasks = [
        dict(name=row[0], due_date=row[1], priority=row[2],
             task_id=row[3]) for row in cursor.fetchall()
    ]
    cursor = g.db.execute(
        'select name, due_date, priority, task_id from tasks where status=0'
    )
    closed_tasks = [
        dict(name=row[0], due_date=row[1], priority=row[2],
             task_id=row[3]) for row in cursor.fetchall()
    ]
    g.db.close()
    return render_template(
        'tasks.html',
        form=AddTaskForm(request.form),
        open_tasks=open_tasks,
        closed_tasks=closed_tasks
    )
```

Make sure to add `g` to the imports:

```
from flask import Flask, flash, redirect, render_template, \
    request, session, url_for, g
```

What's happening?

We queried the database for open and closed tasks and assigned them to two variables, `open_tasks` and `closed_tasks`. We then passed those variables to the `tasks.html` page. These variables will then be used to populate the open and closed task lists, respectively.

Make sense?

Also, you may have noticed this line-

```
form=AddTaskForm(request.form),
```

`AddTaskForm()` will be the name of a form used to, well, add tasks. This has not been created yet.

Add, Update, and Delete Tasks

Next, we need to add the ability to create new tasks, mark tasks as complete, and delete tasks. Add each of these three functions to the `views.py` file:

```
# Add new tasks
@app.route('/add/', methods=['POST'])
@login_required
def new_task():
    g.db = connect_db()
    name = request.form['name']
    date = request.form['due_date']
    priority = request.form['priority']
    if not name or not date or not priority:
        flash("All fields are required. Please try again.")
        return redirect(url_for('tasks'))
    else:
        g.db.execute('insert into tasks (name, due_date, priority, status) \
                     values (?, ?, ?, 1)', [
            request.form['name'],
            request.form['due_date'],
            request.form['priority']
        ])
    g.db.commit()
    g.db.close()
    flash('New entry was successfully posted. Thanks.')
    return redirect(url_for('tasks'))

# Mark tasks as complete
@app.route('/complete/<int:task_id>/')
@login_required
def complete(task_id):
    g.db = connect_db()
    g.db.execute(
        'update tasks set status = 0 where task_id=' + str(task_id)
    )
    g.db.commit()
    g.db.close()
    flash('The task was marked as complete.')
    return redirect(url_for('tasks'))

# Delete Tasks
@app.route('/delete/<int:task_id>/')
@login_required
def delete_entry(task_id):
    g.db = connect_db()
    g.db.execute('delete from tasks where task_id=' + str(task_id))
    g.db.commit()
    g.db.close()
    flash('The task was deleted.')
    return redirect(url_for('tasks'))
```

What's happening?

The last two functions pass in a variable parameter, `task_id`, from the `tasks.html` page (which we will create next). This variable is equal to the unique `task_id` field in the database. A query is then performed and the appropriate action takes place. In this case, an action means either marking a task as complete or deleting a task. Notice how we have to convert the `task_id` variable to a string, since we are using string concatenation to combine the SQL query with the `task_id`, which is an integer.

NOTE: This type of routing is commonly referred to as dynamic routing. Flask makes this easy to implement as we have already seen. Read more about it [here](#).

Tasks Template

Find the `tasks.html` file in the `assests/flasktaskr-01` directory of `book2-exercises` . Copy and paste this code into your own `tasks.html` within your `templates` directory.

Read over the html and see if you can sort out what's going on before continuing.

What's happening?

Although a lot is going on in here, the only thing you have not seen before are these statements:

```
<a href="{{ url_for('delete_entry', task_id = task.task_id) }}">Delete</a>
<a href="{{ url_for('complete', task_id = task.task_id) }}">Mark as Complete</a>
```

Essentially, we pulled the `task_id` from the database dynamically from each row in the database table as the `for` loop iterates. We then assigned the id to a variable, also named `task_id` , which is then passed back to either the `delete_entry()` function -
`@app.route('/delete/<int:task_id>/')` - or the `complete()` function -
`@app.route('/complete/<int:task_id>/')` .

Make sure to walk through this code line by line. You should understand what each line is doing. Add comments to help.

Add Tasks Form

We're now going to use a powerful Flask extension called [WTForms](#) to help with form handling and data validation. Remember how we still need to create the `AddTaskForm()` form? Let's do that now.

First, install the package from the "flasktaskr" directory:

```
$ pip install Flask-WTF==0.12
$ pip freeze > requirements.txt
```

Now let's create a new file called `forms.py`:

```
# project/forms.py

from flask_wtf import Form
from wtforms import StringField, DateField, IntegerField, \
    SelectField
from wtforms.validators import DataRequired

class AddTaskForm(Form):
    task_id = IntegerField()
    name = StringField('Task Name', validators=[DataRequired()])
    due_date = DateField(
        'Date Due (mm/dd/yyyy)',
        validators=[DataRequired()], format='%m/%d/%Y')
    priority = SelectField(
        'Priority',
        validators=[DataRequired()],
        choices=[
            ('1', '1'), ('2', '2'), ('3', '3'), ('4', '4'), ('5', '5'),
            ('6', '6'), ('7', '7'), ('8', '8'), ('9', '9'), ('10', '10')
        ])
    status = IntegerField('Status')
```

Notice how we're importing from both Flask-WTF and WTForms. Essentially, Flask-WTF works in tandem with WTForms, abstracting much of the functionality.

Save the form in the root directory.

What's going on?

As the name suggests, the `validators`, validate the data submitted by the user.

`DataRequired` simply means that the field cannot be blank, while the `format` validator restricts the input to the MM/DD/YY date format.

NOTE: The validators and choices are set up correctly in the form; however, we're not currently using any logic in the `new_task()` view function to prevent a form submission if the submitted data does not conform to the specific validators. We need to use a method called `validate_on_submit()`, which returns `True` if the data passes validation, in the function. We'll look at this further down the road.

Make sure you update your `views.py` by importing the `AddTaskForm()` class from `forms.py`:

```
from forms import AddTaskForm
```

Sanity Check

Finally, test out the functionality of the app.

Fire up your server again. You should be able to log in now. Ensure that you can view tasks, add new tasks, mark tasks as complete, and delete tasks.

If you get any errors, be sure to double check your code.

The screenshot shows the FlaskTaskr application interface. At the top, it says "Welcome to FlaskTaskr" and has a "Logout" link. Below that is a form for "Add a new task" with fields for "Task Name", "Due Date (mm/dd/yyyy)", "Priority" (a dropdown menu showing "1"), and a "Save" button. Underneath this is a section titled "Open tasks:" with a table showing three tasks:

Task Name	Due Date	Priority	Actions
Finish this tutorial	09/22/2016	10	Delete - Mark as Complete
Make some delicious pasta	09/23/2016	10	Delete - Mark as Complete
Install updates and shutdown down my computer	09/24/2016	8	Delete - Mark as Complete

Below the open tasks is a section titled "Closed tasks:" with a table header showing columns for "Task Name", "Due Date", "Priority", and "Actions".

That's it for now. Next time we'll speed up the development process by adding powerful extensions to the application.

Your structure within "project" should look like this:

```
└── _config.py
└── db_create.py
└── flasktaskr.db
└── forms.py
└── run.py
└── static
    ├── css
    │   └── main.css
    ├── img
    └── js
└── templates
    ├── _base.html
    ├── login.html
    └── tasks.html
└── views.py
```

Commit your code to your local repo and then push to Github.

NOTE: If you had any problems with your code or just want to double check your code, be sure to view the *flasktaskr-01* folder in the course [repository](#).

Flask: FlaskTaskr, Part 2 - SQLAlchemy and User Management

Now that we have a functional app, let's add some features and extensions so that the application is easier to develop and manage, as we continue to build out additional functionality. Further, we will also look at how best to structure, test, and deploy the app.

Specifically, we will address:

Task	Complete
Database Management	No
User Registration	No
User Login/Authentication	No
Database Relationships	No
Managing Sessions	No
Error Handling	No
Testing	No
Styling	No
Test Coverage	No
Nose Testing Framework	No
Permissions	No
Blueprints	No
New Features	No
Password Hashing	No
Custom Error Pages	No
Error Logging	No
Deployment Options	No
Automated Deployments	No
Building a REST API	No
Boilerplate and Workflow	No
Continuous Integration and Delivery	No

Before starting this chapter, make sure you can log in with your current app, and then once you're logged in check that you can run all of the CRUD commands against the *tasks* table:

1. **Create** - Add new tasks
2. **Read** - View all tasks
3. **Update** - Mark tasks as "complete"

4. **Delete** - Remove tasks

If not, be sure to grab the code from *flasktaskr-01* from the course [repository](#).

Homework

- Please read over the [main page](#) of the Flask-SQLAlchemy extension. Compare the code samples to regular SQL. How do the classes/objects compare to the SQL statements used for creating a new database table?
- Take a look at all the Flask extensions [here](#). Read them over quickly.

Database Management

As mentioned in the *Database Programming* chapter, you can work with relational databases without having to touch (very much) SQL. Essentially, you need to use an Object Relational Mapper (ORM), which translates and maps SQL commands and your database schema into Python objects. It makes working with relational databases much easier as it eliminates having to write repetitive code.

ORMs also make it easy to switch relational database engines without having to re-write much of the code that interacts with the database itself due to the means in which SQLAlchemy structures the data.

That said, no matter how much you use an ORM you will eventually have to use SQL for troubleshooting or testing quick, one-off queries as well as advanced queries. It's also really, really helpful to know SQL, when trying to decide on the most efficient way to query the database, to know what calls the ORM will be making to the database, and so forth. Learn SQL first, in other words. For more, check out [this](#) popular blog post.

We will be using the [Flask-SQLAlchemy](#) extension, which is a type of ORM, to manage our database.

Let's jump right in.

Note: One major advantage of Flask is that you are not limited to a specific ORM or ODM (Object Document Mapper) for non-relational databases. You can use SQLAlchemy, Peewee, Pony, MongoEngine, etc. The choice is yours.

Setup

Start by installing Flask-SQLAlchemy:

```
$ pip install Flask-SQLAlchemy==2.1
$ pip freeze > requirements.txt
```

Delete your current database, *flasktaskr.db*, and then create a new file called *models.py* in the root directory. We're going to recreate the database using SQLAlchemy. As we do this, compare this method to how we created the database before, using vanilla SQL.

```
# project/models.py

from views import db

class Task(db.Model):

    __tablename__ = "tasks"

    task_id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    due_date = db.Column(db.Date, nullable=False)
    priority = db.Column(db.Integer, nullable=False)
    status = db.Column(db.Integer)

    def __init__(self, name, due_date, priority, status):
        self.name = name
        self.due_date = due_date
        self.priority = priority
        self.status = status

    def __repr__(self):
        return '<name {}>'.format(self.name)
```

We have one class, `Task()`, that defines the `tasks` table. The variable names are used as the column names. *Any field that has a `primary_key` set to `True` will auto-increment.*

Update the imports and configuration section in `views.py`:

```
from forms import AddTaskForm

from functools import wraps
from flask import Flask, flash, redirect, render_template, \
    request, session, url_for
from flask_sqlalchemy import SQLAlchemy

# config

app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)

from models import Task
```

Make sure to remove the following code from `views.py` since we are not using the Python SQLite wrapper to interact with the database anymore:

```
import sqlite3
def connect_db():
    return sqlite3.connect(app.config['DATABASE_PATH'])
```

Update the route handlers in `views.py`:

```
@app.route('/logout/')
def logout():
    session.pop('logged_in', None)
    flash('Goodbye!')
    return redirect(url_for('login'))

@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME'] or \
           request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid Credentials. Please try again.'
            return render_template('login.html', error=error)
        else:
            session['logged_in'] = True
            flash('Welcome!')
            return redirect(url_for('tasks'))
    return render_template('login.html')

@app.route('/tasks/')
@login_required
def tasks():
    open_tasks = db.session.query(Task) \
        .filter_by(status='1').order_by(Task.due_date.asc())
    closed_tasks = db.session.query(Task) \
        .filter_by(status='0').order_by(Task.due_date.asc())
    return render_template(
        'tasks.html',
        form=AddTaskForm(request.form),
        open_tasks=open_tasks,
        closed_tasks=closed_tasks
    )

@app.route('/add/', methods=['GET', 'POST'])
@login_required
def new_task():
    form = AddTaskForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
```

```
new_task = Task(
    form.name.data,
    form.due_date.data,
    form.priority.data,
    '1'
)
db.session.add(new_task)
db.session.commit()
flash('New entry was successfully posted. Thanks.')
return redirect(url_for('tasks'))


@app.route('/complete/<int:task_id>/')
@login_required
def complete(task_id):
    new_id = task_id
    db.session.query(Task).filter_by(task_id=new_id).update({"status": "0"})
    db.session.commit()
    flash('The task is complete. Nice.')
    return redirect(url_for('tasks'))


@app.route('/delete/<int:task_id>/')
@login_required
def delete_entry(task_id):
    new_id = task_id
    db.session.query(Task).filter_by(task_id=new_id).delete()
    db.session.commit()
    flash('The task was deleted. Why not add a new one?')
    return redirect(url_for('tasks'))
```

Pay attention to the differences in the `new_task()` , `complete()` , and `delete_entry()` functions. How are they structured differently from before when we used vanilla SQL for the queries instead?

Also, update the `_config.py` file:

```
# project/_config.py

import os

# grab the folder where this script lives
basedir = os.path.abspath(os.path.dirname(__file__))

DATABASE = 'flasktaskr.db'
USERNAME = 'admin'
PASSWORD = 'admin'
CSRF_ENABLED = True
SECRET_KEY = 'my_precious'

# define the full path for the database
DATABASE_PATH = os.path.join(basedir, DATABASE)

# the database uri
SQLALCHEMY_DATABASE_URI = 'sqlite:////' + DATABASE_PATH
```

Here we're defining the `SQLALCHEMY_DATABASE_URI` to tell SQLAlchemy where to access the database. Confused about `os.path.join` ? Read about it [here](#).

Lastly, update `db_create.py`.

```
# project/db_create.py

from views import db
from models import Task
from datetime import date

# create the database and the db table
db.create_all()

# insert data
db.session.add(Task("Finish this tutorial", date(2016, 9, 22), 10, 1))
db.session.add(Task("Finish Real Python", date(2016, 10, 3), 10, 1))

# commit the changes
db.session.commit()
```

What's happening?

1. We initialize the database schema (i.e., `Task`) by calling `db.create_all()` .

2. We then populate the table with some data, via the `Task` object from `models.py` to specify the schema.
3. To apply the previous changes to our database we need to commit using

```
db.session.commit()
```

Since we are now using SQLAlchemy, we're modifying the way we do database queries. The code is much cleaner. Compare this method to the actual SQL code from the previous chapter.

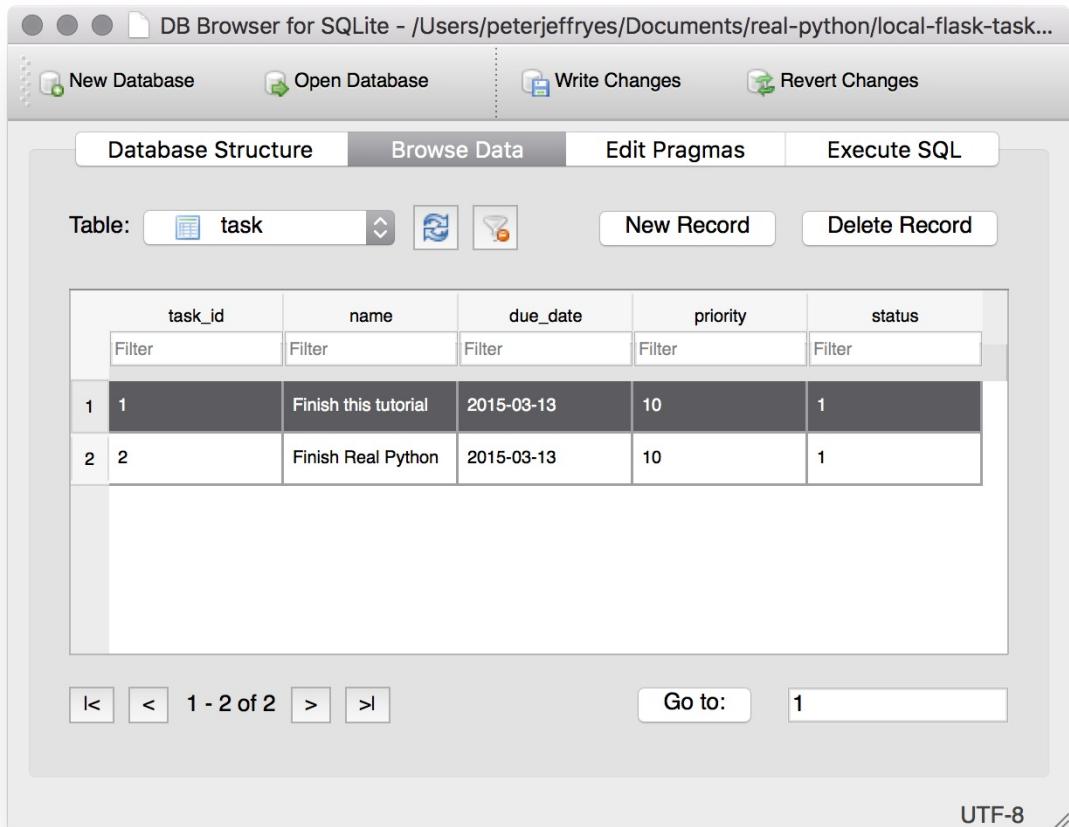
NOTE: Remember when we created the `Task` class? We extended the `db.Model`. This is how the `create_all` command knows to create our `Task` schema.

Create the database

Save all the files, and run the script:

```
$ python db_create.py
```

The `flasktaskr.db` should have been recreated. Open up the file in the SQLite Browser to ensure that the table and the above data are present in the `tasks` table.



CSRF Token

Finally, since we are issuing a POST request, we need to add `{{ form.csrf_token }}` to all forms in the templates. This applies the CSRF prevention setting to the form that we enabled in the configuration.

Place it directly after this line:

```
<form action="{{ url_for('new_task') }}" method="post">
```

The form should now look like this:

```
<form action="{{ url_for('new_task') }}" method="post">
{{ form.csrf_token }}
<td>
<label>Task Name:</label>
<input name="name" type="text">
</td>
<td>
<label>Due Date (mm/dd/yyyy):</label>
<input name="due_date" type="text" width="120px">
</td>
<td>
<label>Priority:</label>
<select name="priority" width="100px">
<option value="1">1</option>
<option value="2">2</option>
<option value="3">3</option>
<option value="4">4</option>
<option value="5">5</option>
<option value="6">6</option>
<option value="7">7</option>
<option value="8">8</option>
<option value="9">9</option>
<option value="10">10</option>
</select>
</td>
<td>
 
 
<input class="button" type="submit" value="Save">
</td>
</form>
```

Test

Fire up the server. Ensure that you can still view tasks, add new tasks, mark tasks as complete, and delete tasks.

Nice. Time to move on to user registration. Don't forget to commit your code and PUSH to Github.

User Registration

Let's allow multiple users to access the task manager by setting up a user registration form.

Create a new table

We need to create a new table in our database to house user data.

To do so, just add a new class to *models.py*:

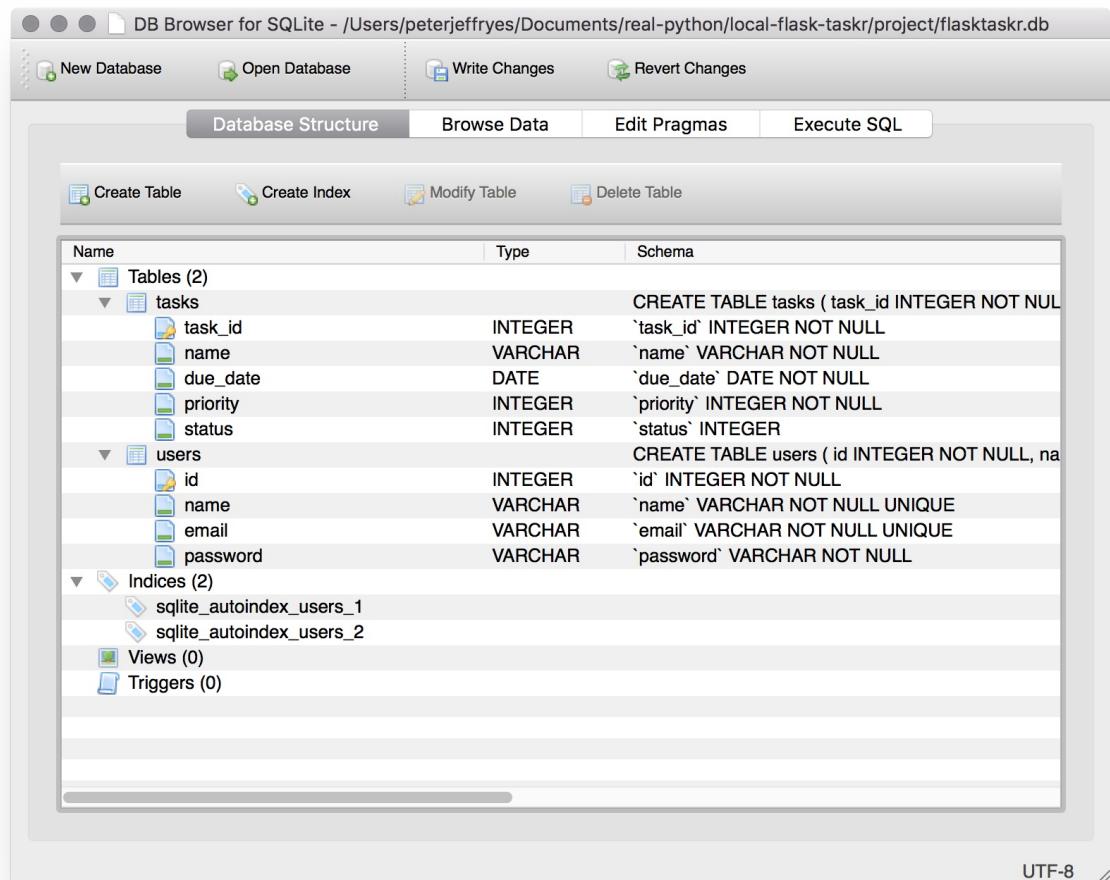
```
class User(db.Model):  
  
    __tablename__ = 'users'  
  
    id = db.Column(db.Integer, primary_key=True)  
    name = db.Column(db.String, unique=True, nullable=False)  
    email = db.Column(db.String, unique=True, nullable=False)  
    password = db.Column(db.String, nullable=False)  
  
    def __init__(self, name=None, email=None, password=None):  
        self.name = name  
        self.email = email  
        self.password = password  
  
    def __repr__(self):  
        return '<User {}>'.format(self.name)
```

Run *db_create.py* again. Before you do so, comment out the following lines:

```
db.session.add(Task("Finish this tutorial", date(2016, 9, 22), 10, 1))  
db.session.add(Task("Finish Real Python", date(2016, 10, 3), 10, 1))
```

If you do not do this, the script will add that data to the database again.

Open up the SQLite Browser. Notice how it ignores the table already created, *tasks*, and just creates the *users* table:



Configuration

Remove the following lines of code in `_config.py`:

```
USERNAME = 'admin'  
PASSWORD = 'admin'
```

We no longer need this configuration since we will use the information from the `users` table in the database instead of the hard-coded data.

We also need to update `forms.py` to cater for both user registration and logging in.

Add the following classes:

```
class RegisterForm(Form):
    name = StringField(
        'Username',
        validators=[DataRequired(), Length(min=6, max=25)])
    email = StringField(
        'Email',
        validators=[DataRequired(), Length(min=6, max=40)])
    password = PasswordField(
        'Password',
        validators=[DataRequired(), Length(min=6, max=40)])
    confirm = PasswordField(
        'Repeat Password',
        validators=[DataRequired(), EqualTo('password', message='Passwords must match')])
class LoginForm(Form):
    name = StringField(
        'Username',
        validators=[DataRequired()])
    password = PasswordField(
        'Password',
        validators=[DataRequired()])
```

Then update the imports:

```
from flask_wtf import Form
from wtforms import StringField, DateField, IntegerField, \
    SelectField, PasswordField
from wtforms.validators import DataRequired, Length, EqualTo
```

Next we need to update the Controller, *views.py*. Update the imports, and add the following code:

```
#####
##### imports #####
#####

from forms import AddTaskForm, RegisterForm, LoginForm

from functools import wraps
from flask import Flask, flash, redirect, render_template, \
    request, session, url_for
from flask_sqlalchemy import SQLAlchemy

#####
##### config #####
#####

app = Flask(__name__)
app.config.from_object('_config')
db = SQLAlchemy(app)

from models import Task, User
```

This allow access to the `RegisterForm()` and `LoginForm()` classes from `forms.py` and the `User()` class from `models.py`.

Add the new view function, `register()`:

```
@app.route('/register/', methods=['GET', 'POST'])
def register():
    error = None
    form = RegisterForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            new_user = User(
                form.name.data,
                form.email.data,
                form.password.data,
            )
            db.session.add(new_user)
            db.session.commit()
            flash('Thanks for registering. Please login.')
            return redirect(url_for('login'))
    return render_template('register.html', form=form, error=error)
```

Here, the user information obtained from the `register.html` template (which we still need to create) is stored inside the variable `new_user`. That data is then added to the database, and after successful registration, the user is redirected to `login.html` with a

message thanking them for registering. `validate_on_submit()` returns either `True` or `False` depending on whether the submitted data passes the form validators associated with each field in the form.

Templates

Go to the [repository](#) and grab the updated html from the `assests/flasktaskr-02` folder.

Save this as `register.html` within your "templates" directory.

Now let's add a registration link to the bottom of the `login.html` page:

```
<br>

<p><em>Need an account? </em><a href="/register">Signup!</a></p>
```

Be sure to remove the following code as well:

```
<p><em>Use 'admin' for the username and password.</em></p>
```

Test it out. Run the server, click the link to register, and register a new user. You should be able to register just fine, as long as the fields pass validation. Everything turn out okay? Double check my code, if not. Now we need to update the code so users can login. Why? Since the logic in the controller is not searching the newly created database table for the correct username and password. Instead, it's still looking for hard-coded values in the `_config.py` file:

```
if request.form['username'] != app.config['USERNAME'] or \
    request.form['password'] != app.config['PASSWORD']:
```

Authentication

The next step for allowing multiple users to login is to change the `login()` function within the controllers as well as the login template.

Controller

Replace the current `login()` function with:

```
@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    form = LoginForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            user = User.query.filter_by(name=request.form['name']).first()
            if user is not None and user.password == request.form['password']:
                session['logged_in'] = True
                flash('Welcome!')
                return redirect(url_for('tasks'))
            else:
                error = 'Invalid username or password.'
        else:
            error = 'Both fields are required.'
    return render_template('login.html', form=form, error=error)
```

This code is not too much different from the old code. When a user submits their user credentials via a POST request, the database is queried for the submitted username and password. If the credentials are not found, an error populates; otherwise, the user is logged in and redirected to `tasks.html`.

Templates

Update `login.html` with the code from the `assests` folder in the [repository](#).

Test it out. Try logging in with the same user you registered. If done correctly, you should be able to log in and then you'll be redirected to `tasks.html`.

Check out the logs in the terminal:

```
127.0.0.1 - - [23/Sep/2016 11:22:30] "POST / HTTP/1.1" 302 -
127.0.0.1 - - [23/Sep/2016 11:22:30] "GET /tasks/ HTTP/1.1" 200 -
```

Can you tell what happened? Can you predict what the logs will look like when you submit a bad username and/or password? Or if you leave a field blank? test it.

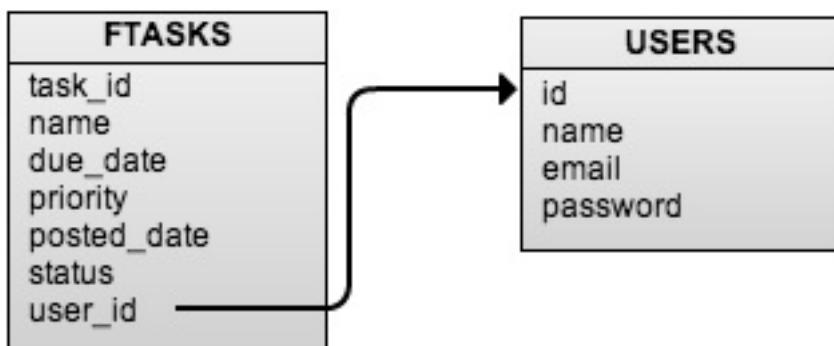
Database Relationships

To complete the conversion to SQLAlchemy we need to update both the database and task template.

First, let's update the database to add two new fields to `tasks` table: `posted_date` and `user_id`. The `user_id` field also needs to link back to the User table.

Database relationships defined

We briefly touched on the subject of relationally linking tables together in the chapter on SQL, but essentially relational databases are designed to connect tables together using unique fields. By linking (or binding) the `id` field from the `users` table with the `user_id` field from the `tasks` table, we can do basic SQL queries to find out who created a certain task as well as find out all the tasks created by a certain user:



Let's look at how to alter the tables to create such relationships within `models.py`.

Add the following field to the "tasks" table-

```
user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
```

-and this field to the "users" table:

```
tasks = db.relationship('Task', backref='poster')
```

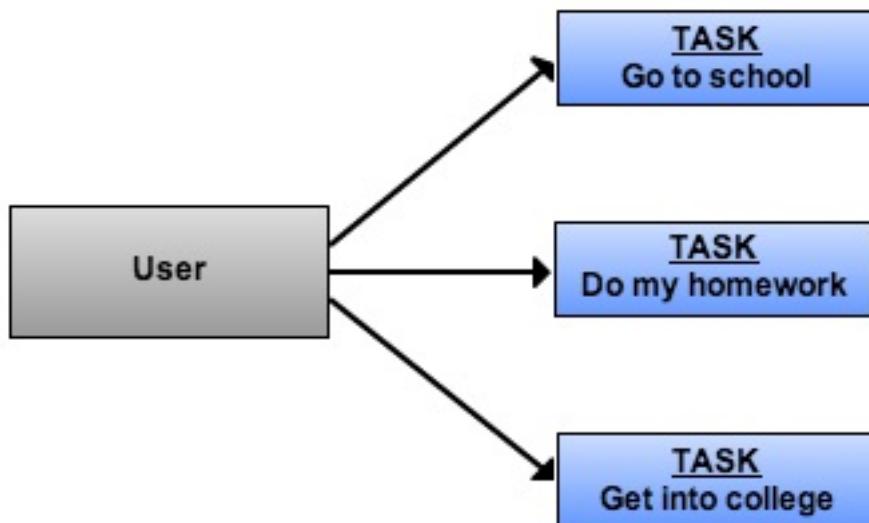
The `user_id` field in the `tasks` table is a foreign key, which binds the values from this field to the values found in the corresponding `id` field in the `users` table. Foreign keys are essential for creating relationships between tables in order to correlate information.

SEE ALSO: Need help with foreign keys? Take a look at the [W3 documentation](#).

Further, in a relational database there are three basic relationships:

1. One to One (1:1) - For example, *one* employee is assigned *one* employee id
2. One to Many (1:M) - *one* department contains *many* employees
3. Many to Many (M:M) - *many* employees take *many* training courses

In our case, we have a one to many relationship: *one* user can post *many* tasks:



If we were to create a more advanced application we could also have a many to many relationship: *many* users could alter *many* tasks. However, we will keep this database simple: one user can create a task, one user can mark that same task as complete, and one user can delete the task.

The `ForeignKey()` and `relationship()` functions are dependent on the type of relationship. In most One to Many relationships the `ForeignKey()` is placed on the "many" side, while the `relationship()` is on the "one" side. The new field associated with the `relationship()` function is not an actual field in the database. Instead, it simply references the objects associated with the "many" side. This can be confusing at first, but it should become clear after you go through an example.

We also need to add another field, "posted_date", to the `Task()` class:

```
# project/models.py

from views import db

import datetime

class Task(db.Model):

    __tablename__ = "tasks"

    task_id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    due_date = db.Column(db.Date, nullable=False)
    priority = db.Column(db.Integer, nullable=False)
    posted_date = db.Column(db.Date, default=datetime.datetime.utcnow())
    status = db.Column(db.Integer)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'))

    def __init__(self, name, due_date, priority, posted_date, status, user_id):
        self.name = name
        self.due_date = due_date
        self.priority = priority
        self.posted_date = posted_date
        self.status = status
        self.user_id = user_id

    def __repr__(self):
        return '<name {}>'.format(self.name)

class User(db.Model):

    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
    password = db.Column(db.String, nullable=False)
    tasks = db.relationship('Task', backref='poster')

    def __init__(self, name=None, email=None, password=None):
        self.name = name
        self.email = email
        self.password = password

    def __repr__(self):
        return '<User {}>'.format(self.name)
```

If we ran the above code, it would only work if we used a fresh, empty database. But since our database already has the "tasks" and "users" tables, SQLAlchemy will not redefine these database tables. To fix this, we need a migration script that will update the schema and transfer any existing data:

```
# project/db_migrate.py

from views import db
from _config import DATABASE_PATH

import sqlite3
from datetime import datetime

with sqlite3.connect(DATABASE_PATH) as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # temporarily change the name of tasks table
    c.execute("""ALTER TABLE tasks RENAME TO old_tasks""")

    # recreate a new tasks table with updated schema
    db.create_all()

    # retrieve data from old_tasks table
    c.execute("""SELECT name, due_date, priority,
                   status FROM old_tasks ORDER BY task_id ASC""")

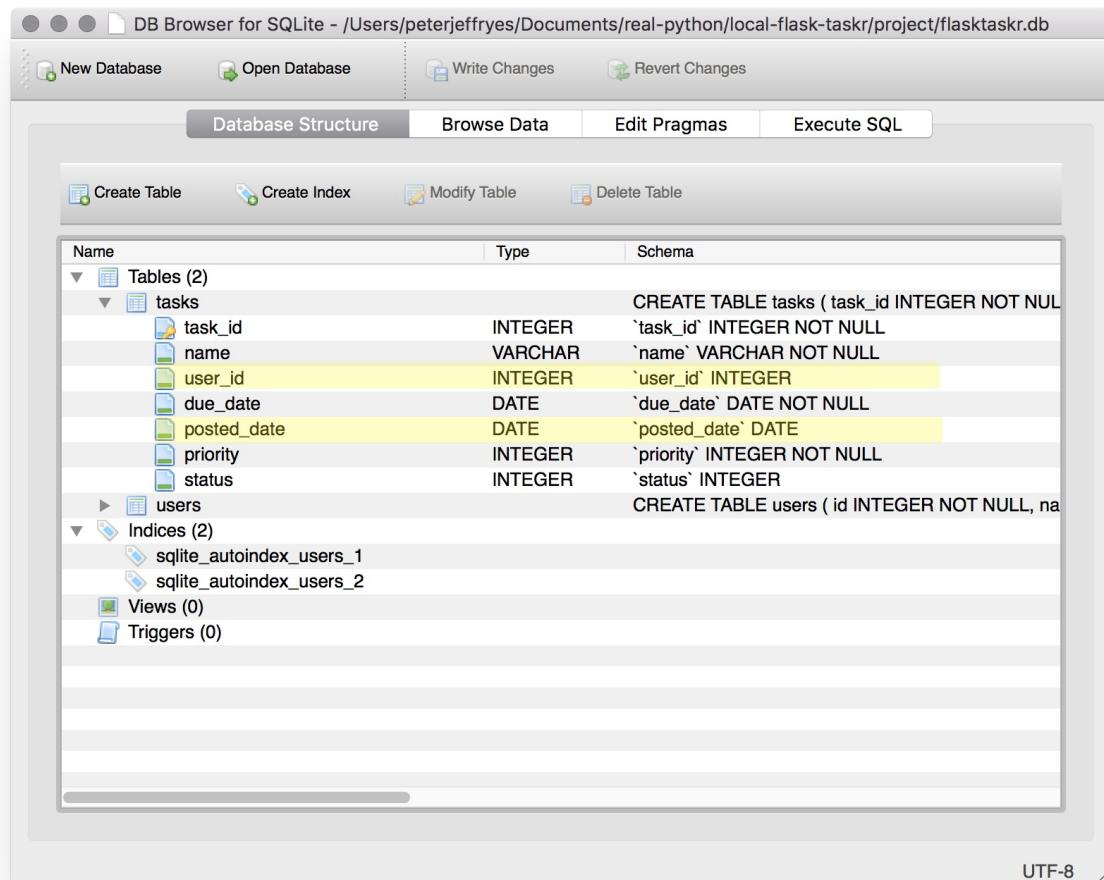
    # save all rows as a list of tuples; set posted_date to now and user_id to 1
    data = [(row[0], row[1], row[2], row[3],
              datetime.now(), 1) for row in c.fetchall()]

    # insert data to tasks table
    c.executemany("""INSERT INTO tasks (name, due_date, priority, status,
                                              posted_date, user_id) VALUES (?,?,?,?,?,?)""", data)

    # delete old_tasks table
    c.execute("DROP TABLE old_tasks")
```

Save this as `db_migrate.py` under the root directory and run it.

Note that this script did not touch the "users" table; it is only the "tasks" table that has underlying schema changes. Using SQLite Browser, verify that the "posted_date" and "user_id" columns have been added to the "tasks" table.



Controller

We also need to update the adding of tasks within `views.py`. Within the `new_task` function, change the following:

```
new_task = Task(
    form.name.data,
    form.due_date.data,
    form.priority.data,
    '1'
)
```

to:

```

new_task = Task(
    form.name.data,
    form.due_date.data,
    form.priority.data,
    datetime.datetime.utcnow(),
    '1',
    '1'
)

```

Make sure you add the following import - `import datetime`.

We added `datetime.datetime.utcnow()` and `'1'`. The former simply captures the current date and passes it to the `Tasks()` class, while the latter assigns `user_id` to 1. This means that any task that we create is owned by/associated with the first user in the `users` database table. This is okay if we only have one user. However, what happens if there is more than one user? Later, in a subsequent section, we will change this to capture the `user_id` of the currently logged-in user.

Templates

Now, let's update the `tasks.html` template. Again head over to the [repository](#) and grab the `tasks.html` out of the `assets/flasktaskr-02` folder.

The changes are fairly straightforward. Can you find them? Take a look at this file along with `forms.py` to see how the drop-down list is implemented.

Now you are ready to test!

Fire up your server and try adding a few tasks. Register a new user and add some more tasks. Make sure that the current date shows up as the posted date. Look back at my code if you're having problems. Also, we can see that the first user is always showing up under *Posted by* - which is expected.

Open tasks:

Task Name	Due Date	Posted Date	Priority	Posted By	Actions
Make some pasta	2016-09-24	2016-09-25	7	michael	Delete - Mark as Complete
Finish This Tutorial	2016-09-24	2016-09-25	10	michael	Delete - Mark as Complete
Restart and Update computer	2016-09-24	2016-09-25	8	michael	Delete - Mark as Complete
Lunch with Sara	2016-09-24	2016-09-25	6	michael	Delete - Mark as Complete
Walk the dog	2016-09-26	2016-09-25	7	michael	Delete - Mark as Complete

Let's correct that.

Add these styles to your `main.css` file:

```
.inline-form .input-group{  
  display: inline-block;  
  margin: 10px 5px;  
}  
  
.input-group input {  
  width: 100%;  
}
```

Managing Sessions

Do you remember the relationship we established between the two tables in the last lesson?

```
user_id = Column(Integer, ForeignKey('users.id'))
tasks = relationship('Task', backref = 'poster')
```

Well, with that simple relationship, we can query for the actual name of the user for each task posted. First, we need to log the `user_id` in the session when a user successfully logs in. So make the following updates to the `login()` function in `views.py`:

```
@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    form = LoginForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            user = User.query.filter_by(name=request.form['name']).first()
            if user is not None and user.password == request.form['password']:
                session['logged_in'] = True
                session['user_id'] = user.id
                flash('Welcome!')
                return redirect(url_for('tasks'))
            else:
                error = 'Invalid username or password.'
        else:
            error = 'Both fields are required.'
    return render_template('login.html', form=form, error=error)
```

Next, when we post a new task, we need to grab the user id from the session and add it to the SQLAlchemy ORM query. So, update the `new_task()` function:

```
@app.route('/add/', methods=['GET', 'POST'])
@login_required
def new_task():
    form = AddTaskForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            new_task = Task(
                form.name.data,
                form.due_date.data,
                form.priority.data,
                datetime.datetime.utcnow(),
                '1',
                session['user_id']
            )
            db.session.add(new_task)
            db.session.commit()
            flash('New entry was successfully posted. Thanks.')
            return redirect(url_for('tasks'))
    else:
        flash('All fields are required.')
        return redirect(url_for('tasks'))
    return render_template('tasks.html', form=form)
```

Here, we grab the current user in session, pulling the `user_id` and adding it to the query.

Another `pop()` method needs to be used for when a user logs out:

```
@app.route('/logout/')
def logout():
    session.pop('logged_in', None)
    session.pop('user_id', None)
    flash('Goodbye!')
    return redirect(url_for('login'))
```

Now open up `tasks.html`. In each of the two for loops, note this statement:

```
<td width="90px">{{ task.poster.name }}</td>
```

Go back to your model. Notice that since we used `poster` as the `backref`, we can use it like a regular query object. Nice!

Fire up your server.

Register a new user and then login using that newly created user. Create a new task and watch how the "Posted By" field now gets populated with the name of the user who created the task.

With that, we're done looking at database relationships as well as the conversion to SQLAlchemy. Again, we can now easily switch SQL database engines (which we will eventually get to). The code now abstracts away much of the repetition from straight SQL so our code is cleaner and more readable.

Next, let's look at form handling as well as unit testing. Take a break, though. You earned it.

Your project structure with the "project" folder should now look like this:

```
└── _config.py
└── db_create.py
└── db_migrate.py
└── flasktaskr.db
└── forms.py
└── models.py
└── run.py
└── static
    ├── css
    │   └── main.css
    ├── img
    └── js
└── templates
    ├── _base.html
    ├── login.html
    ├── register.html
    └── tasks.html
└── views.py
```

If you had any problems with your code or just want to double check your code with mine, be sure to view the *flasktaskr-02* folder in the course [repository](#).

Flask: FlaskTaskr, Part 3 - Error Handling and Testing

Welcome back. Remember where we left off?

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	No
Testing	No
Styling	No
Test Coverage	No
Nose Testing Framework	No
Permissions	No
Blueprints	No
New Features	No
Password Hashing	No
Custom Error Pages	No
Error Logging	No
Deployment Options	No
Automated Deployments	No
Building a REST API	No
Boilerplate and Workflow	No
Continuous Integration and Delivery	No

Let's get to it. First, make sure your app is working properly. Fire up the app. Register a new user, log in, and then add, update, and delete a few tasks. If you come across any problems, compare your code to the code from *flasktaskr-02* in the course [repository](#).

In this section we're going to look at error handling and testing. You will also be introduced to a development strategy known as Test Driven Development.

Error Handling

Errors that occur during execution are called exceptions. Such errors need to be addressed since they stop execution.

Let's look at how to handle one such error...

Form Validation Errors

With the server running, try to register a new user without entering any information, which should result in an exception on the server-side. The problem is that nothing happens on the client-side. Literally. This is confusing for the end user.

We need to add in an error message in order to provide feedback so that the user knows how to proceed. Fortunately, [WTForms](#) can handle this for us for any form that has a validator attached to it.

Open `forms.py`. There are already some validators in place. For example, in the `RegisterForm()` class, the `name` field has a `Length` validator, indicating the inputted value should be between 6 and 25 characters:

```
class RegisterForm(Form):
    name = StringField(
        'Username',
        validators=[DataRequired(), Length(min=6, max=25)])
    email = StringField(
        'Email',
        validators=[DataRequired(), Length(min=6, max=40)])
    password = PasswordField(
        'Password',
        validators=[DataRequired(), Length(min=6, max=40)])
    confirm = PasswordField(
        'Repeat Password',
        validators=[DataRequired(), EqualTo('password', message='Passwords must match')])
```

Let's add a few more validators:

```
class RegisterForm(Form):
    name = StringField(
        'Username',
        validators=[DataRequired(), Length(min=6, max=25)])
    email = StringField(
        'Email',
        validators=[DataRequired(), Email(), Length(min=6, max=40)])
    password = PasswordField(
        'Password',
        validators=[DataRequired(), Length(min=6, max=40)])
    confirm = PasswordField(
        'Repeat Password',
        validators=[DataRequired(), EqualTo('password')])
)
```

Be sure to update the imports as well:

```
from wtforms.validators import DataRequired, Length, EqualTo, Email
```

Now we need to display the error messages to the end user. To do so, simply add the following code to the `views.py` file:

```
def flash_errors(form):
    for field, errors in form.errors.items():
        for error in errors:
            flash(u"Error in the %s field - %s" % (
                getattr(form, field).label.text, error), 'error')
```

Then update the templates...

- `register.html`
- `login.html`
- `tasks.html`

You can find the updated html in `assests/flasktaskr-03` in the course [repository](#).

NOTE: Instead of labels, notice how we're using placeholders. This is purely an aesthetic change. If it does not suite you, you can always change it back.

Add the CSS styles associated with the `error` class:

```
.error {  
    color: red;  
    font-size: .8em;  
    background-color: #FFFFFF;  
}
```

Update the view as well:

```
@app.route('/add/', methods=['GET', 'POST'])  
@login_required  
def new_task():  
    error = None  
    form = AddTaskForm(request.form)  
    if request.method == 'POST':  
        if form.validate_on_submit():  
            new_task = Task(  
                form.name.data,  
                form.due_date.data,  
                form.priority.data,  
                datetime.datetime.utcnow(),  
                '1',  
                session['user_id'])  
            db.session.add(new_task)  
            db.session.commit()  
            flash('New entry was successfully posted. Thanks.')  
            return redirect(url_for('tasks'))  
        else:  
            return render_template('tasks.html', form=form, error=error)  
    return render_template('tasks.html', form=form, error=error)
```

Did this work? Fire up the server again. Try inputting invalid values in the form. The proper error messages should display. Did you notice that the open and closed tasks don't show up? This is because our logic in the `new_task` route renders the template without the tasks if we encounter an error.

To update that, create helper functions out of the opening and closing of tasks.

```
def open_tasks():  
    return db.session.query(Task).filter_by(  
        status='1').order_by(Task.due_date.asc())  
  
def closed_tasks():  
    return db.session.query(Task).filter_by(  
        status='0').order_by(Task.due_date.asc())
```

Now update `tasks()` and `new_task()` :

```
@app.route('/tasks/')
@login_required
def tasks():
    return render_template(
        'tasks.html',
        form=AddTaskForm(request.form),
        open_tasks=open_tasks(),
        closed_tasks=closed_tasks()
    )

@app.route('/add/', methods=['GET', 'POST'])
@login_required
def new_task():
    error = None
    form = AddTaskForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            new_task = Task(
                form.name.data,
                form.due_date.data,
                form.priority.data,
                datetime.datetime.utcnow(),
                '1',
                session['user_id']
            )
            db.session.add(new_task)
            db.session.commit()
            flash('New entry was successfully posted. Thanks.')
            return redirect(url_for('tasks'))
    return render_template(
        'tasks.html',
        form=form,
        error=error,
        open_tasks=open_tasks(),
        closed_tasks=closed_tasks()
    )
```

Try this out again.

Database Related Errors

What happens when you try to register a new user with a username (or email address) that already exists in the database? You should see an ugly `IntegrityError`. We can use a `try/except` to handle the error.

First add another import to `views.py`:

```
from sqlalchemy.exc import IntegrityError
```

Then update the `register()` function:

```
@app.route('/register/', methods=['GET', 'POST'])
def register():
    error = None
    form = RegisterForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            new_user = User(
                form.name.data,
                form.email.data,
                form.password.data,
            )
            try:
                db.session.add(new_user)
                db.session.commit()
                flash('Thanks for registering. Please login.')
                return redirect(url_for('login'))
            except IntegrityError:
                error = 'That username and/or email already exist.'
                return render_template('register.html', form=form, error=error)
    return render_template('register.html', form=form, error=error)
```

Essentially, the code within the `try` block attempts to execute. If it fails due to the exception specified in the `except` block, the code execution stops and the code within the `except` block is ran. If the error does not occur then the program fully executes and the `except` block is skipped altogether.

Try registering a user again with a username that already exists. The error should be handled correctly now.

NOTE: You will *never* be able to anticipate every error in your code. Error handlers (when used right) help to catch common errors so that they are handled gracefully.

Testing

Testing your application is an absolute necessity, especially as your app grows in size and complexity. Tests help ensure that as the complexity of an application grows the various moving parts continue to work together in a harmonious fashion. In other words, tests help reveal when code isn't working correctly and when it breaks altogether.

Every time you add a feature to an application, fix a bug, or change some code you should make sure the code, new and old, is adequately covered by tests and that the tests all pass after you're done.

Test Driven Development

[Test Driven Development](#) is a form of software development where a test is written before any code. This helps you fully hash out your application's requirements. It also prevents over-coding: You develop your application until the test passes, adding no more code than necessary. Keep in mind though that it's difficult, if not impossible, to establish all your test cases beforehand. You also do not want to limit creativity, so be careful with being overly attached to the notion of writing every single test case first. Give yourself permission to explore and be creative.

Getting Started

We will use the unit test framework `unittest`, which is part of the Python standard library, to write our tests.

The structure is simple: Each test case is written as a separate method within a larger class.

```
import unittest

class TestCase(unittest.TestCase):

    # place your test methods here

    if __name__ == '__main__':
        unittest.main()
```

You can break classes into several test suites. For example, one suite could test the managing of users and sessions, while another could test user registration, and so forth. Such test suites are meant to affirm that the desired outcome does not deviate from the actual outcome.

Let's create a base test script:

```
# project/test.py

import os
import unittest

from views import app, db
from _config import basedir
from models import User

TEST_DB = 'test.db'

class AllTests(unittest.TestCase):

    #####
    ##### setup and teardown #####
    #####
    # executed prior to each test
    def setUp(self):
        app.config['TESTING'] = True
        app.config['WTF_CSRF_ENABLED'] = False
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///\\' + \
            os.path.join(basedir, TEST_DB)
        self.app = app.test_client()
        db.create_all()

    # executed after each test
    def tearDown(self):
        db.session.remove()
        db.drop_all()

    # each test should start with 'test'
    def test_user_setup(self):
        new_user = User("michael", "michael@mherman.org", "michaelherman")
        db.session.add(new_user)
        db.session.commit()

    if __name__ == "__main__":
        unittest.main()
```

Save it as `test.py` in the root directory and run it:

```
$ python project/test.py
.
-----
Ran 1 test in 0.223s

OK
```

It passed!

What happened?

1. The `setUp()` method was invoked which created a test database (if it did not already exist) and applied the database schema from the main database. It also created a `test client`, which handles requests and sends back responses for us to test. It essentially mocks out the entire Flask app.
2. The `test_user_setup()` method was called, inserting data into the "users" table.
3. Lastly, the `tearDown()` method was invoked which dropped all the tables in the test database.

Try commenting out the `tearDown()` method and run the test script once again. Check the database in the SQLite Browser. Is the data there? While the `tearDown()` method is still commented out, run the test script a second time.

You should see:

```
sqlalchemy.exc.IntegrityError: (IntegrityError) UNIQUE constraint failed: users.em
ail 'INSERT INTO users (name, email, password) VALUES (?, ?, ?)' ('michael', 'mich
ael@herman.org', 'michaelherman')

-----
Ran 1 test in 0.044s

FAILED (errors=1)
```

What happened?

An exception was thrown because the username and email address was not unique (as defined in our `user()` class in `models.py`). Delete `test.db` and add the `tearDown()` method back in before moving on.

Assert

Each test should have an `assert()` method to either verify an expected result or a condition or indicate that an exception is raised.

Let's quickly look at an example of how `assert()` works. Update `test.py` with the following code:

```
# project/test.py

import os
import unittest

from views import app, db
from _config import basedir
from models import User

TEST_DB = 'test.db'

class AllTests(unittest.TestCase):

    #####
    ##### setup and teardown #####
    #####
    # executed prior to each test
    def setUp(self):
        app.config['TESTING'] = True
        app.config['WTF_CSRF_ENABLED'] = False
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///\\' + \
            os.path.join(basedir, TEST_DB)
        self.app = app.test_client()
        db.create_all()

    # executed after each test
    def tearDown(self):
        db.session.remove()
        db.drop_all()

    # each test should start with 'test'
    def test_users_can_register(self):
        new_user = User("michael", "michael@mherman.org", "michaelherman")
        db.session.add(new_user)
        db.session.commit()
        test = db.session.query(User).all()
        for t in test:
            t.name
        assert t.name == "michael"

    if __name__ == "__main__":
        unittest.main()
```

In this example, we're testing whether a new user is successfully added to the database. We then pull all the data from the database, `test = db.session.query(User).all()` , extract just the name, and then test to make sure the name equals the expected result -

which is "michael".

Run the test suite in the current form. It should pass. Then change the assert statement to -

```
assert t.name != "michael"
```

Now you can see what an assertion error looks like:

```
=====
FAIL: test_users_can_register (__main__.AllTests)
-----
Traceback (most recent call last):
  File "test.py", line 41, in test_users_can_register
    assert t.name != "michael"
AssertionError

-----
Ran 1 test in 0.044s

FAILED (failures=1)
```

Change the assert statement back to `assert t.name == "michael"`. Run the tests again to make sure nothing broke.

What are we testing?

Think about this test for a minute. What exactly are we testing? That the data was inserted correctly into the database, right?. Put another way, we're ensuring that the SQLAlchemy library works as expected by making sure the new user was added to the database. This type of test is called an integration test. If you go back to the code, try to find all the separate functions this test actually covers. There are several. If this test breaks in the future, you will then have to manually test each function to isolate the problem unless you have tests set up to assert that each of those functions work. Those tests - which test individual functions - are called unit test, since they test a single unit of code.

Semantics aside, this is not a course on testing. Testing is an art. It takes years and years of practice to hone this skill. This course teaches you *how* to test, and then, over time, you'll learn *what* to test. It's your decision whether you want to write more granular unit tests or higher-level integration tests on your apps going forward.

Current Functionality

Let's test the our app's functionality up until this point, function by function. For now, let's add all of our tests to the `test.py` file. We'll refactor this later after the tests work.

Where should we begin?

Take out a piece of paper and review the code in the `views.py` file, making a note of everything that can be tested. You want *at least* enough tests to cover each function. Try to break this down between users and tasks.

Function	Not logged in	Logged in	What are we testing?
<code>login()</code>	X	X	Form is present page
<code>login()</code>	X	-	Unregistered users cannot login
<code>login()</code>	X	-	Registered users can log in (form validation)
<code>register()</code>	X	-	Form is present on the register page
<code>register()</code>	X	-	Users can register (form validation)
<code>logout()</code>	-	X	Users can log out
<code>tasks()</code>	-	X	Users can access tasks
<code>tasks()</code>	-	X	Users can add tasks (form validation)
<code>tasks()</code>	-	X	Users can complete tasks
<code>tasks()</code>	-	X	Users can delete tasks

Let's go through and start testing, one by one. Make sure to run your tests after each new test is added to avoid code regressions.

Homework

- Although not specifically about Flask, watch [this](#) excellent video on testing Python code.

Users

Form is present

```
def test_form_is_present(self):
    response = self.app.get('/')
    self.assertEqual(response.status_code, 200)
    self.assertIn(b'Please sign in to access your task list', response.data)
```

This test catches the response from sending a GET request to '/' and then asserts that the response static code is 200 and that the words 'Please sign in to access your task list' are present.

Why do you think we went the extra mile to test that specific HTML is present?

If we just tested for a 200 response code, we don't know what the end user actually sees. It could be JSON or an entirely different HTML page.

NOTE: In the `assertIn` syntax we added a `b` before the text that we are looking for. When using the `assertIn` method, `unittest` is looking for a *bytes-like object* and this `b` converts the string to the appropriate type.

Unregistered users cannot log in

First, let's use a helper method to log users in as necessary. Since we'll have to do this multiple times throughout the test suit it makes sense to abstract the code out into a separate method, and then call it as necessary.

```
def login(self, name, password):
    return self.app.post('/', data=dict(
        name=name, password=password), follow_redirects=True)
```

And here's the test:

```
def test_users_cannot_login_unless_registered(self):
    response = self.login('foo', 'bar')
    self.assertIn(b'Invalid username or password.', response.data)
```

Registered users can log in (form validation)

Here, we need to test the form validation. On the database level, since we are using SQLAlchemy, data that contains special characters is automatically escaped, helping to prevent injection attacks. So, as long as you use the normal mechanism for adding data

to the database, then you can be confident that your app is protected against dangerous injections. To be safe, we can test the schema, starting with valid data.

Add another helper method to register a user:

```
def register(self, name, email, password, confirm):
    return self.app.post(
        'register/',
        data=dict(name=name, email=email, password=password, confirm=confirm),
        follow_redirects=True
    )
```

Test that the user can log in:

```
def test_users_can_login(self):
    self.register('Michael', 'michael@realpython.com', 'python', 'python')
    response = self.login('Michael', 'python')
    self.assertIn(b'Welcome!', response.data)
```

Finally, let's test some bad data and see how far the process gets

```
def test_invalid_form_data(self):
    self.register('Michael', 'michael@realpython.com', 'python', 'python')
    response = self.login('alert("alert box!");', 'foo')
    self.assertIn(b'Invalid username or password.', response.data)
```

This is a very similar test to `test_users_cannot_login_unless_registered()`. Perhaps they can be combined?

Form is present on register page

```
def test_form_is_present_on_register_page(self):
    response = self.app.get('register/')
    self.assertEqual(response.status_code, 200)
    self.assertIn(b'Please register to access the task list.', response.data)
```

This should be straightforward. Maybe we should test that the actual form is preset rather than just some other element on the same page as the form?

Users can register (form validation)

```
def test_user_registration(self):
    self.app.get('register/', follow_redirects=True)
    response = self.register(
        'Michael', 'michael@realpython.com', 'python', 'python')
    self.assertIn(b'Thanks for registering. Please login.', response.data)
```

Look back at your tests. We're already testing for this, right? Should we refactor?

```
def test_user_registration_error(self):
    self.app.get('register/', follow_redirects=True)
    self.register('Michael', 'michael@realpython.com', 'python', 'python')
    self.app.get('register/', follow_redirects=True)
    response = self.register(
        'Michael', 'michael@realpython.com', 'python', 'python')
    self.assertIn(
        b'That username and/or email already exist.',
        response.data)
```

This code works, but is it DRY?

Users can log out

Let's test to make sure that *only* logged in users can log out. In other words, if you're not logged in, you should be redirected to the homepage.

Start by adding another helper method:

```
def logout(self):
    return self.app.get('logout/', follow_redirects=True)
```

Now we can test logging out for both logged in and not logged in users:

```
def test_logged_in_users_can_logout(self):
    self.register('Fletcher', 'fletcher@realpython.com', 'python101', 'python101')
    self.login('Fletcher', 'python101')
    response = self.logout()
    self.assertIn(b'Goodbye!', response.data)

def test_not_logged_in_users_cannot_logout(self):
    response = self.logout()
    self.assertNotIn(b'Goodbye!', response.data)
```

Run the tests. You should get a failure since users not logged in can still access that endpoint, `/logout`. To fix that simply add the `@login_required` decorator to the view:

```
@app.route('/logout')
@login_required
def logout():
    session.pop('logged_in', None)
    session.pop('user_id', None)
    flash('Goodbye!')
    return redirect(url_for('login'))
```

Test it again. All should pass.

Users can access tasks

```
def test_logged_in_users_can_access_tasks_page(self):
    self.register(
        'Fletcher', 'fletcher@realpython.com', 'python101', 'python101'
    )
    self.login('Fletcher', 'python101')
    response = self.app.get('tasks/')
    self.assertEqual(response.status_code, 200)
    self.assertIn(b'Add a new task:', response.data)

def test_not_logged_in_users_cannot_access_tasks_page(self):
    response = self.app.get('tasks/', follow_redirects=True)
    self.assertIn(b'You need to login first.', response.data)
```

Seems like you could combine the last two tests, right?

Tasks

For these next set of tests assume that only users that are logged in can add, complete, or delete tasks. Why? We already know that only logged in users can access the 'tasks/' endpoint. No need to test that again.

Also, lets add two more helper methods:

```
def create_user(self, name, email, password):
    new_user = User(name=name, email=email, password=password)
    db.session.add(new_user)
    db.session.commit()

def create_task(self):
    return self.app.post('add/', data=dict(
        name='Go to the bank',
        due_date='10/08/2016',
        priority='1',
        posted_date='10/08/2016',
        status='1'
    ), follow_redirects=True)
```

Users can add tasks (form validation)

```
def test_users_can_add_tasks(self):
    self.create_user('Michael', 'michael@realpython.com', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    response = self.create_task()
    self.assertIn(
        b'New entry was successfully posted. Thanks.', response.data
    )
```

What if there's an error?

```
def test_users_cannot_add_tasks_when_error(self):
    self.create_user('Michael', 'michael@realpython.com', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    response = self.app.post('add/', data=dict(
        name='Go to the bank',
        due_date='',
        priority='1',
        posted_date='02/05/2014',
        status='1'
    ), follow_redirects=True)
    self.assertIn(b'This field is required.', response.data)
```

Users can complete tasks

```
def test_users_can_complete_tasks(self):
    self.create_user('Michael', 'michael@realpython.com', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    response = self.app.get("complete/1/", follow_redirects=True)
    self.assertIn(b'The task is complete. Nice.', response.data)
```

Users can delete tasks

```
def test_users_can_delete_tasks(self):
    self.create_user('Michael', 'michael@realpython.com', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    response = self.app.get("delete/1/", follow_redirects=True)
    self.assertIn(b'The task was deleted.', response.data)
```

Did we miss anything?

Remember when we set up the 'users' table and defined the relationship between users and tasks, establishing a one-to-many relationship: *one user can create a task, one user can mark that same task as complete, and one user can delete the task?* If user A adds a task then that task can only be updated or deleted by user A can update and/or delete that task.

Let's add a test for that:

```
def test_users_cannot_complete_tasks_that_are_not_created_by_them(self):
    self.create_user('Michael', 'michael@realpython.com', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    self.logout()
    self.create_user('Fletcher', 'fletcher@realpython.com', 'python101')
    self.login('Fletcher', 'python101')
    self.app.get('tasks/', follow_redirects=True)
    response = self.app.get("complete/1/", follow_redirects=True)
    self.assertNotIn(
        b'The task is complete. Nice.', response.data
    )
```

By now you probably already know that test is going to fail, just from manual testing alone. We need to write the code to get it to pass. This marks the beginning of Test Driven Development! We'll pick up here next time.

Your project structure should now look like this:

```
└── _config.py
└── db_create.py
└── db_migrate.py
└── flasktaskr.db
└── forms.py
└── models.py
└── run.py
└── static
    ├── css
    │   └── main.css
    ├── img
    └── js
└── templates
    ├── _base.html
    ├── login.html
    ├── register.html
    └── tasks.html
└── test.db
└── test.py
└── views.py
```

NOTE: The code for this chapter can be found in the *flasktaskr-03* folder in the course [repository](#)

Interlude: Intro to HTML and CSS

Let's take a break from Flask to cover HTML and CSS...

This is a two part tutorial covering HTML, CSS, JavaScript, and jQuery, where we will be building a basic todo list. In part one, the focus is on HTML and CSS.

NOTE: This is a beginner tutorial. If you already have a basic understanding of HTML and CSS, feel free to skip.

Webpages are made up of many things, but HTML (Hyper Text Markup Language) and CSS (Cascading Style Sheets) are two of the most important components. Together, they are the building blocks for every single page on the Internet.

Think of a car. It, too, is made up of many attributes. Doors. Windows. Tires. Seats. In the world of HTML, these are the `elements` of a webpage. Meanwhile, each of the car's attributes are usually different. Perhaps they differ by size. Or color. Or wear and tear. These attributes are used to define how the `elements` look. Back in the world of web, CSS is used to define the look and feel of a webpage.

Now let's turn to an actual web page...

HTML

HTML provides structure, making it viewable by a web browser.

To start, let's add some basic structure. Copy and paste the below structure into your text editor. Save the file as *index.html* in a new directory called "front-end".

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

This structure is commonly referred to as a boilerplate template. Such templates are used to speed up the development process so you don't have to code the features common to every single webpage each time you create a new page. Most boilerplates include much more structure, but let's start with the basics for now.

What's going on?

1. The first line, `<!DOCTYPE html>` is the document type declaration, which tells the browser that HTML5 is used. Without this, browsers can get confused, especially older versions of Internet Explorer.
2. `<html>` is the first tag and it informs the browser that all code between the opening and closing, `</html>`, tags is HTML.
3. The `<head>` tag contains links to CSS stylesheets and JavaScript files that as well as meta information used by search engines for classification.
4. All code that falls within the `<body>` tags are part of the main content of the page, which will appear in the browser to the end user.

This is how a standard HTML page, following HTML5 standards, is structured.

Moving on, add four new tags:

1. `title` `<title>`
2. `heading` `<h1>`
3. `break` `
`
4. `paragraph` `<p>`

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Todo List</title>
  </head>
  <body>
    <h1>My Todo List</h1>
    <p>Get yourself organized!</p>
    <br>
  </body>
</html>
```

Elements, Tags, and Attributes

1. Tags form the structure of your page. They surround and apply *meaning* to content. There usually is an opening tag and then a closing tag, like - `<h1></h1>` , a heading. Some tags, like the `
` (line break) tag do not require a closing tag.
2. Elements represent the tags as well as whatever falls between the opening and closing tags, like - `<title>My Todo List</title>`
3. Attributes are key-value pairs that provide additional information to an element. For example, `id` s or `class` es (sometimes referred to as selectors) are used to select a particular tag for either applying JavaScript or CSS styles.

SEE ALSO: Mozilla has an excellent reference guide for all [HTML elements](#) for all HTML elements.

Additional Tags

Let's add some more tags:

```
<!doctype html>
<html>
  <head>
    <title>My Todo List</title>
  </head>
  <body>
    <h1>My Todo List</h1>
    <p>Get yourself organized!</p>
    <br>
    <form>
      <input type="text" placeholder="Enter a todo... ">
    </form>
    <br>
    <button>Submit!</button>
    <br>
  </body>
</html>
```

We added:

1. A form (`<form>`), with one input, for entering a todo.
2. A button (`<button>`) that's used for submitting the entered todo.

Check it out in your browser. Kind of bland, right? Fortunately, we can quickly change that with CSS!

CSS

While HTML provides structure, CSS is used for styling. From the size of the text to the background colors to the positioning of HTML elements, CSS gives you control over almost every visual aspect of a page.

CSS and HTML work in tandem. CSS styles are applied directly to HTML elements, as you will soon see.

NOTE: There are three ways that you can assign styles to HTML tags - inline, internal, and external. Inline styles are placed directly in an HTML tag. Internal styles fall within the head of the HTML. These should be avoided, as it's best practice to separate HTML and CSS (don't mix structure with presentation!).

First, we need to "link" the HTML page and CSS stylesheet. Add the following code to the `<head>` section of the HTML page just above the title:

```
<link rel="stylesheet" type="text/css" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
<link rel="stylesheet" type="text/css" href="main.css">
```

Let's also add a `container` class:

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="http://netdna.bootstrapcdn.com/
      bootswatch/3.0.3/flatly/bootstrap.min.css">
    <link rel="stylesheet" type="text/css" href="main.css">
    <title>My Todo List</title>
  </head>
  <body>
    <div class="container">
      <h1>My Todo List</h1>
      <p>Get yourself organized!</p>
      <br>
      <form>
        <input type="text" placeholder="Enter a todo...">
      </form>
      <br>
      <button>Submit!</button>
      <br>
    </div>
  </body>
</html>
```

Save the file. The first CSS file is a [bootstrap](#) stylesheet, while the second is a custom stylesheet, which we will create in a few moments. For more information on Bootstrap, please the [Getting Started with Bootstrap 3](#) blog post.

Open the page in your web browser. See the difference? Yes, it's subtle - but the font is different along with the style of the input boxes. What styles does the [container](#) class add?

Custom Styles

Create a *main.css* file and save it in the same folder as your *index.html* file. Then add the following CSS to the file:

```
.container {
  max-width: 500px;
  padding-top: 50px;
}
```

Save. Refresh *index.html* in your browser.

What's going on?

Look back at the CSS file.

1. We have the `.container` *selector*, which is associated with the selector in our HTML document, followed by curly braces.
2. Inside the curly braces, we have *properties*, which are descriptive phrases, like `font-weight` , `font-size` , OR `background-color` .
3. *Values* are then assigned to each property, which are preceded by a colon and followed by a semi-colon. [CSS Values](#) is an excellent resource for finding the acceptable values given a CSS property. Bookmark it!

Putting it all together

Add some attributes to your HTML:

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="http://netdna.bootstrapcdn.com/
      bootstrap/3.0.3/flatly/bootstrap.min.css">
    <link rel="stylesheet" type="text/css" href="main.css">
    <title>My Todo List</title>
  </head>
  <body>
    <div class="container">
      <h1>My Todo List</h1>
      <p class="lead">Get yourself organized!</p>
      <br>
      <form id="my-form" role="form">
        <input id="my-input" class="form-control" type="text" placeholder="Enter a
        todo...">
      </form>
      <br>
      <button class="btn btn-primary btn-md">Submit!</button>
      <br>
    </div>
  </body>
</html>
```

Do you see the selectors? Look for the new `id` s and `class` es. The `id` s will all be used for JavaScript (in the next part of this tutorial), while the `class` es are all Bootstrap styles. If you're curious, check out the [Bootstrap page](#) to see more info about the classes used.

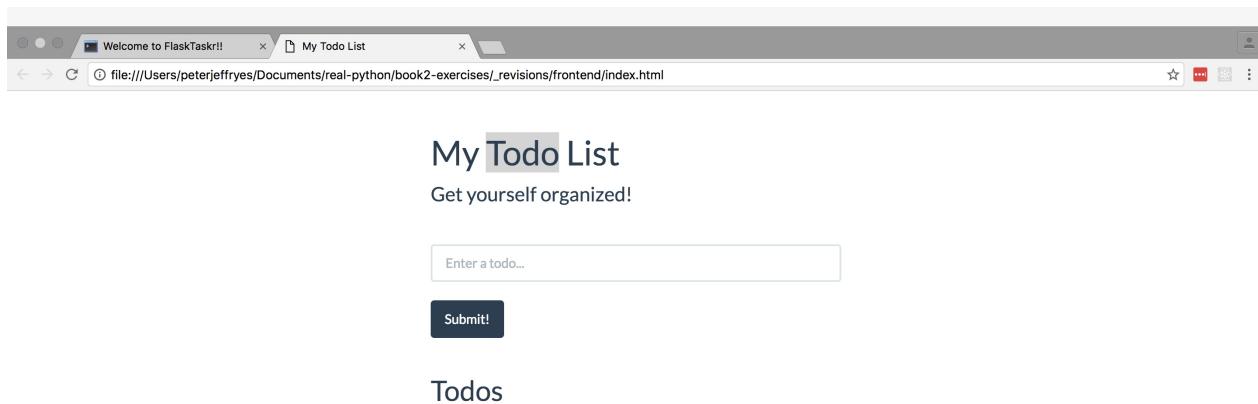
Save. Refresh your browser.

What do you think? Good? Bad? Ugly? Make any additional changes that you'd like.

Chrome Developer Tools

Using Chrome Developer Tools, we can test temporary changes to either HTML or CSS directly from the browser. This can save a lot time, plus you can make edits to any webpage, not just your own.

Open up the HTML page we worked on. Right Click on the heading. Select "Inspect Element".



The screenshot shows a browser window with the title "Welcome to FlaskTaskr!!" and the tab "My Todo List". The URL in the address bar is "file:///Users/peterjeffries/Documents/real-python/book2-exercises/_revisions/frontend/index.html". The page content is "My Todo List" and "Get yourself organized!". Below is a text input field with placeholder "Enter a todo..." and a "Submit!" button. The developer tools pane is open, showing the "Elements" tab with the HTML structure of the page. The heading "My Todo List" is selected, and its styles are displayed in the "Styles" panel on the right, including Bootstrap's Lato font and color #2c3e50.

Todos



The screenshot shows the Chrome Developer Tools Elements pane. The left sidebar shows the HTML structure of the "Todos" page. The right pane shows the "Styles" tab with the computed styles for the selected heading. The styles include "font-family: 'Lato', 'Helvetica Neue', Helvetica, Arial, sans-serif;" and "color: #2c3e50;". The "Computed" tab shows the final styles after Bootstrap's "bootstrap.min.css" is applied. The "Event Listeners" tab shows no listeners for the selected element.

Notice the styles on the right side of the Developer Tools pane associated with the heading. You can change them directly from that pane. Try adding the following style:

```
color: red;
```

This should change the color of the heading to red. Check out the live results in your browser. You can also edit your HTML in real-time. With Dev Tools open, right click the paragraph text in the left pane, select "Edit as HTML", and then add another paragraph.

WARNING: Be careful as these changes are temporary. Watch what happens when you refresh the page. Poof!

Again, this is a great way to test temporary HTML and CSS changes live in your browser. You can also debug and learn how to imitate a desired HTML, CSS, or JavaScript effect from a different webpage.

Make sure both your *.html* and *.css* files are saved.

In the second tutorial we'll add user interactivity with JavaScript and jQuery so that we can actually add and remove todo items.

Homework

- If you want some extra practice, go through the Codecademy series on [HTML and CSS](#).
- Want even more practice with CSS? Try [CSS Diner](#).

Flask: FlaskTaskr, Part 4 - Styles, Test Coverage, and Permissions

Back to *FlaskTaskr*. Fire up your virtual environment and then run the tests:

```
.....F.  
=====
```

FAIL: test_users_cannot_complete_tasks_that_are_not_created_by_them (`__main__.AllTests`)

```
-----  
-----  
-----  
Ran 17 tests in 0.661s  
  
FAILED (failures=1)
```

Right where we left off. Before we write the code to get that test to pass, let's put some of your new HTML and CSS skills to use!

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	Yes
Unit Testing	Yes
Styling	No
Test Coverage	No
Nose Testing Framework	No
Permissions	No
Blueprints	No
New Features	No
Password Hashing	No
Custom Error Pages	No
Error Logging	No
Deployment Options	No
Automated Deployments	No
Building a REST API	No
Boilerplate and Workflow	No
Continuous Integration and Delivery	No

Templates and Styling

Now that we're done with creating the basic app, let's update the styles. Thus far, we've been using the CSS styles from the main Flask tutorial. Let's add our own styles. We'll start with using [Bootstrap](#) to add a basic design template, then we'll edit our CSS file to make style changes to the template.

Bootstrap is a front-end framework that makes your app look good right out of the box. You can just use the generic styles; however, it's **best** to make some changes so that the layout doesn't look like a cookie-cutter template. The framework is great. You'll get the essential tools (mostly CSS and HTML but some Javascript as well) needed to build a nice-looking website at your disposal. As long as you have a basic understanding of HTML and CSS, you can create a design quickly.

You can either download the associated files - [Bootstrap](#) and [jQuery](#) - and place them in your project directory:

```
└ static
  └ css
    └ bootstrap.min.css
  └ js
    └ jquery-1.11.3.min.js
    └ bootstrap.min.js
  └ styles.css
```

Or you can just link directly to the styles in your `_base.html` file via a public content delivery network (CDN), which is a repository of commonly used files.

Either method is fine. Let's use the latter method.

NOTE If you think there will be times where you'll be working on your app without Internet access then you should use the former method just to be safe. Your call.

Parent Template

Add the following files to `_base.html`:

```
<!-- styles -->
<link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/yeti/bootstrap.min.css" rel="stylesheet">

<!-- scripts -->
<script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js"></script>
```

Now add some bootstrap styles. Update your `_base.html` with the html in the `assets` folder of the [project repository](#).

Without getting into too much detail, we just pulled in the Bootstrap stylesheets, added a navigation bar to the top, and used the bootstrap classes to style the app. Be sure to check out the bootstrap [documentation](#) for more information as well as [this](#) blog post.

If it helps, compare the current template to the code we had before the changes. Have a look at the `flasktaskr-03` version in the [repo](#).

Fire up the app and take a look. See the difference. Now, let's update the child templates.

Again, go into the [repo](#) and grab the `flasktaskr-04` version of the `login.html`, `register.html`, `tasks.html` from the `assets` folder.

Custom Styles

Update the styles in `main.css`. Remove everything in there and replace it with these four classes. Bootstrap is doing the rest.

```
body {  
  padding: 60px 0;  
  background: #ffffff;  
}  
  
.footer {  
  padding-top: 30px;  
}  
  
.error {  
  color: red;  
  font-size: .8em;  
}  
  
.input-group {  
  margin: 15px 0;  
}
```

Continue to make as many changes as you'd like. Make it unique. See what you can do on your own. Show it off. Email it to us at info@realpython.com to share your app.

Before moving on, let's run the test suite:

```
$ python project/test.py
```

You should still see one error. Before we address it though, let's install Coverage.

Test Coverage

The best way to reduce bugs and ensure working code is to have a comprehensive test suite in place. Working in tandem with unit testing, Coverage testing is a great way to achieve this as it analyzes your code base and returns a report showing the parts not covered by a test. Keep in mind that even if 100% of your code is covered by tests, there still could be [issues](#) due to flaws in how you structured your tests.

To implement coverage testing, we'll use a tool called [coverage](#).

IMPORTANT: In this section all commands are written assuming that your working directory is *root/project/*.

Install:

```
$ pip install coverage==4.2
$ pip freeze > requirements.txt
```

To run coverage, use the following command:

```
$ coverage run test.py
```

To get a command line coverage report run:

```
$ coverage report --omit=../env/*
```

Name	Stmts	Miss	Cover

_config.py	7	0	100%
forms.py	17	0	100%
models.py	33	2	94%
test.py	120	0	100%
views.py	89	4	96%

TOTAL	266	6	98%

NOTE: `--omit=../env/*` excludes all files within the virtual environment

To print a more robust HTML report:

```
$ coverage html --omit=../env/*
```

Check the report in the newly created "htmlcov" directory by opening *index.html* in your browser.

If you click on one of the modules, you'll see the actual results, line by line. Lines highlighted in red are currently not covered by a test.

One thing stands out from this:

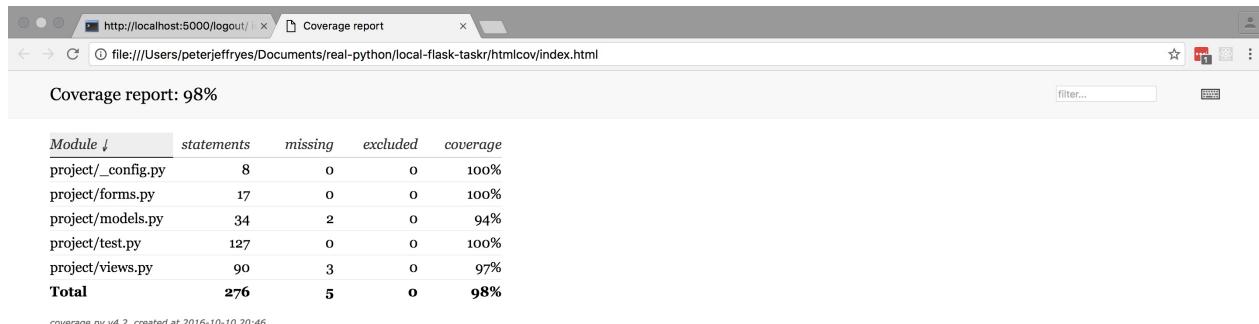
We do not need to explicitly handle form errors in the `/` endpoint - `error = 'Both fields are required.'` - since errors are handled by the form. So we can remove these lines of code from the view:

```
else:  
    error = 'Both fields are required.'
```

Re-run coverage:

```
$ coverage run test.py  
$ coverage html --omit=../env/*
```

Our coverage for the views should have jumped from 95% to 97%. Nice.



Module	statements	missing	excluded	coverage
project/_config.py	8	0	0	100%
project/forms.py	17	0	0	100%
project/models.py	34	2	0	94%
project/test.py	127	0	0	100%
project/views.py	90	3	0	97%
Total	276	5	0	98%

coverage.py v4.2, created at 2016-10-10 20:46

NOTE: Make sure to add both `.coverage` and "htmlcov" to the `.gitignore` file. These should not be part of your repository.

NOTE: If you get stuck with coverage, try calling `coverage erase` and starting over. Each time you run `coverage run test.py`, coverage remembers and is basing its calculations on the state of the test suite at that point. Changes to the code may make it necessary to erase coverage and start fresh.

Nose Testing Framework

[Nose](#) is a powerful utility used for test discovery. Regardless of where you place your tests in your project, Nose will find them. Simply pre-fix your test files with `test_` along with the methods inside the test files. It's that simple. Since we have been following best practices with regard to naming and structuring tests, Nose should just work.

Well, we do need to install it first:

```
$ pip install nose==1.3.7
$ pip freeze > requirements.txt
```

Running Nose

Now you should be able to run Nose, along with coverage:

```
$ nosetests --with-coverage --cover-erase --cover-package=../project
```

Let's look at the flags that we used:

1. `--with-coverage` checks for test coverage.
2. `--cover-erase` erases coverage results from the previous test run.
3. `--cover-package=../project` specifies which portion of the code to analyze.

Run the tests:

Name	Stmts	Miss	Cover
<hr/>			
<code>_config</code>	7	0	100%
<code>db_create.py</code>	5	5	0%
<code>db_migrate.py</code>	12	12	0%
<code>forms.py</code>	17	0	100%
<code>models.py</code>	33	2	94%
<code>test.py</code>	120	0	100%
<code>views.py</code>	88	3	97%
<hr/>			
<code>TOTAL</code>	282	23	98%
<hr/>			
Ran 17 tests <code>in</code> 1.904s			
FAILED (failures=1)			

Homework

- Refactor the entire test suite. We're duplicating code in a number of places. Also, split the tests into two files - `test_users.py` and `test_tasks.py`. Please note: This is a complicated assignment, which you are not expected to get *right*. In fact, there are a number of issues with how these tests are setup that should be addressed, which we will look at later. For now, focus on splitting up the tests, making sure that the split does not negatively effect coverage, as well as eliminating duplicate code.

NOTE: Going forward, the tests are based on a number of refactors. Be sure to do this homework and then compare your solution/code to mine found in the [repo](#). Do your best to refactor on your own and tailor the code found in subsequent sections to reflect the changes based on **your** refactors. If you get stuck, feel free to use/borrow/steal the code from the repo, just make sure you understand it first.

Permissions

To address the final failing test,

`test_users_cannot_complete_tasks_that_are_not_created_by_them()` , we need to add user permissions into the mix. There's really a few different routes we could take.

First, we could use a robust extension like [Flask-Principal](#). Or we could build our own solution. Think about what we need to accomplish? Do we really need to build a full permissions solution?

Probably not. Right now we're not even really dealing with permissions. We just need to throw an error if the user trying to update or delete a task is not the user that added the task. We can add that functionality with an if statement. Thus, something like Flask-Principal is too much for our needs right now. If this is an app that you want to continue to build, perhaps you should add a more robust solution. That's beyond the scope of this course though.

For now, we'll start by adding the code to get our test to pass and then implement a basic permissions feature to allow users with the role of 'admin' to update and delete *all* posts, regardless of whether they posted them or not.

NOTE If you completed the last homework assignment, you should have two test files, `test_users.py` and `test_tasks.py`. Please double check the code in the [repo](#) to ensure that we're on the same page.

Users can only update tasks that they created

Start by running the tests:

```
$ nosetests
```

As expected, we have one failure:

```
$ nosetests
.....
=====
FAIL: test_users_cannot_complete_tasks_that_are_not_created_by_them (test_tasks.TasksTests)
-----
-----
Ran 21 tests in 1.029s

FAILED (failures=1)
```

Update the code

```
@app.route('/complete/<int:task_id>/')
@login_required
def complete(task_id):
    new_id = task_id
    task = db.session.query(Task).filter_by(task_id=new_id)
    if session['user_id'] == task.first().user_id:
        task.update({"status": "0"})
        db.session.commit()
        flash('The task is complete. Nice.')
        return redirect(url_for('tasks'))
    else:
        flash('You can only update tasks that belong to you.')
        return redirect(url_for('tasks'))
```

Here, we're querying the database for the row associated with the `task_id`, as we did before. However, instead of just updating the status to `0`, we're checking to make sure that the `user_id` associated with that specific task is the same as the `user_id` of the user in session.

Run the tests

```
$ nosetests
.....
-----
Ran 21 tests in 0.942s

OK
```

Yay!

Refactor

Before declaring a victory, let's update the test to check for the flashed message:

```
def test_users_cannot_complete_tasks_that_are_not_created_by_them(self):
    self.create_user('Michael', 'michael@realpython.com', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    self.logout()
    self.create_user('Fletcher', 'fletcher@realpython.com', 'python101')
    self.login('Fletcher', 'python101')
    self.app.get('tasks/', follow_redirects=True)
    response = self.app.get("complete/1/", follow_redirects=True)
    self.assertNotIn(
        b'The task is complete. Nice.', response.data
    )
    self.assertIn(
        b'You can only update tasks that belong to you.', response.data
    )
```

Run your tests again. They should still pass! Now, let's do the same thing for deleting tasks ...

Users can only delete tasks that they created

Write the test

```
def test_users_cannot_delete_tasks_that_are_not_created_by_them(self):
    self.create_user('Michael', 'michael@realpython.com', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    self.logout()
    self.create_user('Fletcher', 'fletcher@realpython.com', 'python101')
    self.login('Fletcher', 'python101')
    self.app.get('tasks/', follow_redirects=True)
    response = self.app.get("delete/1/", follow_redirects=True)
    self.assertIn(
        b'You can only delete tasks that belong to you.', response.data
    )
```

Add this code to the `test_tasks.py` file.

Run the tests

The new test will fail, which is expected.

Update the code

Finally, let's write just enough code to get this test to pass. You probably guessed that the refactored `delete_entry()` function will be very similar to the `complete()` function that we refactored last time. Regardless, notice how we are actually specifying how our app should function by writing the test first. Now we just need to write that new functionality.

```
@app.route('/delete/<int:task_id>/')
@login_required
def delete_entry(task_id):
    new_id = task_id
    task = db.session.query(Task).filter_by(task_id=new_id)
    if session['user_id'] == task.first().user_id:
        task.delete()
        db.session.commit()
        flash('The task was deleted. Why not add a new one?')
        return redirect(url_for('tasks'))
    else:
        flash('You can only delete tasks that belong to you.')
        return redirect(url_for('tasks'))
```

Run the tests...

And they pass!

Admin Permissions

Finally, let's add two roles to the database - `user` and `admin`. The former will be the default, and if a user has a role of `admin` they can update or delete any tasks.

Again, start with a test

Add the following code to the `test_users.py` file:

```
def test_default_user_role(self):  
  
    db.session.add(  
        User(  
            "Johnny",  
            "john@doe.com",  
            "johnny"  
        )  
    )  
  
    db.session.commit()  
  
    users = db.session.query(User).all()  
    print(users)  
    for user in users:  
        self.assertEqual(user.role, 'user')
```

Run the tests. Fail.

Write the code

Remember what we have to do in order to update the database without losing any data?

Run a migration. Turn back to *FlaskTaskr (part 2)* to see how we did this.

Basically, we need to *update the model*, *update the migration script*, and then *run the migration script*.

Update the model:

```
class User(db.Model):  
  
    __tablename__ = 'users'  
  
    id = db.Column(db.Integer, primary_key=True)  
    name = db.Column(db.String, unique=True, nullable=False)  
    email = db.Column(db.String, unique=True, nullable=False)  
    password = db.Column(db.String, nullable=False)  
    tasks = db.relationship('Task', backref='poster')  
    role = db.Column(db.String, default='user')  
  
    def __init__(self, name=None, email=None, password=None, role=None):  
        self.name = name  
        self.email = email  
        self.password = password  
        self.role = role  
  
    def __repr__(self):  
        return '<User {}>'.format(self.name)
```

Update the migration script:

```
# project/db_migrate.py  
  
from views import db  
# from datetime import datetime  
from _config import DATABASE_PATH  
import sqlite3  
  
# with sqlite3.connect(DATABASE_PATH) as connection:  
  
#     # get a cursor object used to execute SQL commands  
#     c = connection.cursor()  
  
#     # temporarily change the name of tasks table  
#     c.execute("""ALTER TABLE tasks RENAME TO old_tasks""")  
  
#     # recreate a new tasks table with updated schema  
#     db.create_all()  
  
#     # retrieve data from old_tasks table  
#     c.execute("""SELECT name, due_date, priority,  
#               status FROM old_tasks ORDER BY task_id ASC""")  
  
#     # save all rows as a list of tuples; set posted_date to now and user_id to 1  
#     data = [(row[0], row[1], row[2], row[3],  
#              datetime.now(), 1) for row in c.fetchall()]
```

Permissions

```
#     # insert data to tasks table
#     c.executemany("""INSERT INTO tasks (name, due_date, priority, status,
#                           posted_date, user_id) VALUES (?, ?, ?, ?, ?, ?)""", data)

#     # delete old_tasks table
#     c.execute("DROP TABLE old_tasks")

with sqlite3.connect(DATABASE_PATH) as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # temporarily change the name of users table
    c.execute("""ALTER TABLE users RENAME TO old_users""")

    # recreate a new users table with updated schema
    db.create_all()

    # retrieve data from old_users table
    c.execute("""SELECT name, email, password
                FROM old_users
                ORDER BY id ASC""")

    # save all rows as a list of tuples; set role to 'user'
    data = [(row[0], row[1], row[2],
              'user') for row in c.fetchall()]

    # insert data to users table
    c.executemany("""INSERT INTO users (name, email, password,
                                         role) VALUES (?, ?, ?, ?)""", data)

    # delete old_users table
    c.execute("DROP TABLE old_users")
```

Run the migration:

```
$ python db_migrate.py
```

If all goes well you shouldn't get any errors. Make sure the migration worked. Open the SQLite Database browser. You should see all the same users plus an additional 'role' column.

Run the tests. Pass.

Start with a test (or two)

So, now that users are associated with a role, let's update the logic in our `complete()` and `delete_entry()` functions to allow users with a role of `admin` to be able to update and delete *all* tasks.

Add the following tests to `test_tasks.py`:

```
def test_admin_users_can_complete_tasks_that_are_not_created_by_them(self):
    self.create_user()
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    self.logout()
    self.create_admin_user()
    self.login('Superman', 'allpowerful')
    self.app.get('tasks/', follow_redirects=True)
    response = self.app.get("complete/1/", follow_redirects=True)
    self.assertNotIn(
        'You can only update tasks that belong to you.', response.data
    )

def test_admin_users_can_delete_tasks_that_are_not_created_by_them(self):
    self.create_user()
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    self.logout()
    self.create_admin_user()
    self.login('Superman', 'allpowerful')
    self.app.get('tasks/', follow_redirects=True)
    response = self.app.get("delete/1/", follow_redirects=True)
    self.assertNotIn(
        'You can only delete tasks that belong to you.', response.data
    )
```

Then add a new helper method:

```
def create_admin_user(self):
    new_user = User(
        name='Superman',
        email='admin@realpython.com',
        password='allpowerful',
        role='admin'
    )
    db.session.add(new_user)
    db.session.commit()
```

What happens when you run the tests?

Write the code

This is a fairly easy fix.

Update the login and logout functionality:

```

@app.route('/logout')
@login_required
def logout():
    session.pop('logged_in', None)
    session.pop('user_id', None)
    session.pop('role', None)
    flash('Goodbye!')
    return redirect(url_for('login'))


@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    form = LoginForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            user = User.query.filter_by(name=request.form['name']).first()
            if user is not None and user.password == request.form['password']:
                session['logged_in'] = True
                session['user_id'] = user.id
                session['role'] = user.role
                flash('Welcome!')
                return redirect(url_for('tasks'))
            else:
                error = 'Invalid username or password.'
    return render_template('login.html', form=form, error=error)

```

Here, we're simply adding the user's role to the session cookie on the login, then removing it on logout.

Update the `complete()` and `delete_entry()` functions by simply updating the conditional for both:

```
if session['user_id'] == task.first().user_id or session['role'] == "admin":
```

Now the if `user_id` in session matches the `user_id` that posted the task or if the user's role is 'admin', then that user has permission to update or delete the task.

Retest:

```
$ nosetests
.
.
.
-----
Ran 25 tests in 1.252s

OK
```

Awesome.

Next time

This brings us to a nice stopping point. When we start back up, we'll look at restructuring our app using a modular design pattern called [Blueprints](#).

Homework

- Be sure to read over the official documentation on [Blueprints](#). Cheers!

Flask: FlaskTaskr, Part 5 - Blueprints

Last time we added new CSS styles, code coverage, and app permissions:

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	Yes
Testing	Yes
Styling	Yes
Test Coverage	Yes
Nose Testing Framework	Yes
Permissions	Yes
Blueprints	No
New Features	No
Password Hashing	No
Custom Error Pages	No
Error Logging	No
Deployment Options	No
Automated Deployments	No
Boilerplate and Workflow	No
Building a REST API	No
Continuous Integration and Delivery	No

As promised, this section looks at [Blueprints](#). Before we start, be sure to:

1. Read about the benefits of using Blueprints from the official Flask [documentation](#).
2. Run the test suite again, making sure all tests pass.
3. Ensure all dependencies have been added to *requirements.txt*: `pip freeze > requirements.txt`.
4. Commit the code to your local repo.

What are Blueprints?

[Blueprints](#) provide a means of breaking up a large application into separate, distinct components, each with their own views, templates, and static files.

For a blog, you could have a Blueprint for handling user authentication, another could be used to manage posts, and one more could provide a robust admin panel. Each Blueprint is really a separate app, each handling a different function. Designing your app in this manner significantly increases overall maintainability and re-usability by encapsulating code, templates, and static files/media. This, in turn, decreases development time as it makes it easier for developers to find mistakes, so less time is spent on fixing bugs.

NOTE: Before moving on, please note that this is not a complicated lesson but there are a number of layers to it so it can be confusing. Programming in general is nothing more than combining various layers of knowledge on top of one another. Take it slow. Read through the lesson once without touching any code. Read. Take notes. Draw diagrams. Then when you're ready, go through it again and refactor your app. This is a manual process that will only make sense to you if you first understand the "what and why" before diving in.

Example Code

Let's look at a sample Blueprint for the management of posts for a blog. `admin` , `posts` , and `users` represent separate apps.

What are Blueprints?

```
└── _config.py
└── run.py
└── blog
    ├── __init__.py
    ├── admin
    │   ├── static
    │   ├── templates
    │   └── views.py
    ├── models.py
    ├── posts
    │   ├── static
    │   ├── templates
    │   └── views.py
    └── users
        ├── static
        ├── templates
        └── views.py
```

Let's take a quicker look at the `posts` Blueprint:

```
# blog/posts/views.py

from flask import Blueprint, render_template, url_for

from _config import db
from forms import PostForm
from models import Post

# create a blueprint object
post_blueprint = Blueprint('blog_posts', __name__,)

@post_blueprint.route("/")
def read_posts():
    posts = Post.query.all()
    return render_template('posts.html', posts=posts)

@post_blueprint.route("/add/", methods=['GET', 'POST'])
def create_posts():
    form = PostForm()
    if form.validate_on_submit():
        new_post = Post(form.title.data, form.description.data)
        db.session.add(new_post)
        db.session.commit()
        flash('Post added!')
        return redirect(url_for('read_posts'))
    return render_template('blog/add.html', form=form)
```

What are Blueprints?

Here, a new Blueprint is initialized that gets assigned to the `blueprint` variable. Each view function is bound with the `@blueprint.route`, which let's Flask know that each function is available for use within the entire app.

Again, the above code defined the Blueprint, now we need to register it with the main Flask object:

```
# blog/init.py

from flask import Flask
from posts.views import post_blueprint

blueprint = Flask(__name__)
blueprint.register_blueprint(post_blueprint)
```

That's it.

Refactoring

Now, let's convert *FlaskTaskr* over to the Blueprint pattern.

Step 1: Planning

First, we need to determine how we should logically divide up the app. The simplest way is by functionality:

1. **Users**: handles user log in/log out, registration, and authentication
2. **Tasks**: handles the CRUD functionality associated with the tasks resource

Next, we need to determine what the new directory structure will look like. What does that mean exactly? Well, we need a folder for each Blueprint, "users" and "tasks", each containing:

```
└── __init__.py
└── forms.py
└── static
└── templates
└── views.py
```

The current structure looks like this:

```
└── _config.py
└── db_create.py
└── db_migrate.py
└── flasktaskr.db
└── forms.py
└── models.py
└── run.py
└── static
    ├── css
    │   └── main.css
    ├── img
    └── js
└── templates
    ├── _base.html
    ├── login.html
    ├── register.html
    └── tasks.html
└── test.db
└── test_tasks.py
└── test_users.py
└── views.py
```

Thus, we'll probably want the following structure when all is said and done:

```

├── db_create.py
├── db_migrate.py
└── project
    ├── __init__.py
    ├── _config.py
    ├── flasktaskr.db
    ├── models.py
    └── static
        ├── css
        │   └── main.css
        ├── img
        └── js
    ├── tasks
    │   ├── __init__.py
    │   ├── forms.py
    │   └── views.py
    ├── templates
    │   ├── _base.html
    │   ├── login.html
    │   ├── register.html
    │   └── tasks.html
    ├── test.db
    └── users
        ├── __init__.py
        ├── forms.py
        └── views.py
└── requirements.txt
└── run.py
└── tests
    ├── test_tasks.py
    └── test_users.py

```

Study this new structure. Take note of the difference between the files and folders within each of the Blueprints and the project root.

Let's walk through the conversion process together.

Step 2: Reorganizing the structure

1. Within the "project" directory, add the Blueprint directories, "tasks" and "users".
2. Add two empty `__init__.py` files within "users" and "tasks", which indicate to the Python interpreter that those directories should be treated as modules. Add one to the "project" directory as well.
3. Add "static" and "templates" directories within both the "tasks" and "users" directories.
4. Repeat the previous two steps, creating the files and folders, within the "project"

- directory.
5. Add a "tests" directory outside the "project" directory, and then move both `test_tasks.py`, *and* `test_users.py`* to the new directory.
 6. Then move the following files from outside the project directory - `db_create.py`, `db_migrate.py`, and `run.py`.

You should now have this:

```
└── db_create.py
└── db_migrate.py
└── project
    ├── __init__.py
    ├── _config.py
    ├── forms.py
    ├── models.py
    ├── static
    │   ├── css
    │   │   └── main.css
    │   ├── img
    │   └── js
    ├── tasks
    │   └── __init__.py
    ├── templates
    │   ├── _base.html
    │   ├── login.html
    │   ├── register.html
    │   └── tasks.html
    ├── users
    │   └── __init__.py
    └── views.py
└── run.py
└── tests
    ├── test_tasks.py
    └── test_users.py
```

Step 3: Creating the Blueprints

We now need to create the Blueprints by moving relevant code from `views.py` and `forms.py` found in the root to each Blueprint. The goal is simple: Each Blueprint should have the views and forms associated with that specific Blueprint.

Start by creating new `views.py` and `forms.py` in each Blueprint folder.

Users Blueprint

Views

Pull up the `flasktaskr-05/users/views.py` file from the inside `assests` folder within the [exercises repo](#) and cut and paste it into you project. Have a look at the code.

What's going on here?

We defined the `users` Blueprint and bound each function with the `@users_blueprint.route` decorator so that when we register the Blueprint, Flask will recognize each of the functions.

Forms

Back to the `assets` folder in the [repo](#), `flasktaskr-05/users/forms.py`. Again, cut and paste this into you project.

Tasks Blueprint

We'll do the same for the tasks blueprint.

Views

Pick out the differences between this code and the original task crud methods.

Forms

You know what to do :)

tasks.html

Last one! Keep the `tasks.html` in `project/templates`. Update it with the code in the `assests` folder.

PAUSE

Before continuing, be sure you have thoroughly compared the new and old code. Understanding the difference is essential.

Step 4: Updating the `main` app

With the Blueprints done, let's now turn our attention to the main project, where we still need to register each module (blueprint).

`__init__.py`

```
# project/__init__.py

from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_pyfile('_config.py')
db = SQLAlchemy(app)

from project.users.views import users_blueprint
from project.tasks.views import tasks_blueprint

# register our blueprints
app.register_blueprint(users_blueprint)
app.register_blueprint(tasks_blueprint)
```

run.py

```
# run.py

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

Update the Models

You need to update the import within *models.py*.

From:

```
from views import db
```

To:

```
from project import db
```

Step 5: Testing

Since we moved a number of files around, your virtual environment may not hold the correct requirements. To be safe, remove the virtual environment, then and create a new one. Be sure that you have the most recent dependencies written to your

requirements.txt file. If you are not sure run the `freeze` command (remember the syntax?) before deleting your virtual environment.

```
$ deactivate
$ rm -rf env/
$ python3 -m venv env
$ source env/bin/activate
$ pip install -r requirements.txt
```

Now run the app:

```
$ python run.py
```

If all went well, you shouldn't get any errors. Make sure you can load the main page.

Step 6: Creating the new Database

Let's start fresh by deleting the old database and creating a new one.

Remove the *flasktaskr.db* from the *project* folder.

Update *db_create.py*

```
# db_create.py

from datetime import date

from project import db
from project.models import Task, User

# create the database and the db table
db.create_all()

# insert data
db.session.add(
    User("admin", "ad@min.com", "admin", "admin")
)
db.session.add(
    Task("Finish this tutorial", date(2015, 3, 13), 10, date(2015, 2, 13), 1, 1)
)
db.session.add(
    Task("Finish Real Python", date(2015, 3, 13), 10, date(2015, 2, 13), 1, 1)
)

# commit the changes
db.session.commit()
```

Run the script:

```
$ python db_create.py
```

Make sure to check the database in the SQLite Browser to ensure all the data was added properly.

Step 7: Testing (redux)

Let's manually test again. Fire up the server. Check every link, register a new user, log in and log out. Then check the CRUD functionality - create, update, and delete a number of tasks. Everything should work the same as before. All done! Again, by breaking our application up logically by responsibility, our code is cleaner and more readable. As the app grows, it will be much easier to develop the additional functionalities due to the separation of concerns. This is just the tip of the iceberg; there's many more advanced approaches for using Blueprints that are beyond the scope of this course. Be sure to check out the official Flask [docs](#) for more info.

NOTE: If you had trouble following the restructuring, please view the "flasktaskr-05" directory in the exercises [repo](#).

Finally, let's run some tests.

Make sure to update the imports in `test_tasks.py` and `test_users.py`:

```
import os
import unittest

from project import app, db
from project._config import basedir
from project.models import Task, User
```

Now run the tests:

```
$ nosetests --with-coverage --cover-erase --cover-package=project
.
.
.
Name           Stmts   Miss  Cover
-----
project.py        9      0  100%
project._config.py    7      0  100%
project.models.py     35      2  94%
project.tasks.py      0      0  100%
project.tasks.forms.py  9      0  100%
project.tasks.views.py 55      0  100%
project.users.py      0      0  100%
project.users.forms.py 11      0  100%
project.users.views.py 50      0  100%
-----
TOTAL           176      2  99%
-----
Ran 25 tests in 1.679s
OK
```

Flask: FlaskTaskr, Part 6 - New features and Error Handling

Where are we at with the app?

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	Yes
Testing	Yes
Styling	Yes
Test Coverage	Yes
Nose Testing Framework	Yes
Permissions	Yes
Blueprints	Yes
New Features	No
Password Hashing	No
Custom Error Pages	No
Error Logging	No
Deployment Options	No
Automated Deployments	No
Building a REST API	No
Continuous Integration and Delivery	No

Alright. Let's add some new features utilizing test driven development.

New Features

Display User Name

Here we just want to display the logged in user's name on the task page, on the right side of the navigation bar.

Test

```
def test_task_template_displays_logged_in_user_name(self):
    self.register(
        'Fletcher', 'fletcher@realpython.com', 'python101', 'python101'
    )
    self.login('Fletcher', 'python101')
    response = self.app.get('tasks/', follow_redirects=True)
    self.assertIn(b'Fletcher', response.data)
```

Here we assert that the rendered HTML contains the logged in user's name. Run your tests to watch this one fail. Now write *just enough* code to get it to pass.

Code

First, let's add the user's `name` to the session during the logging in process in `users/views.py`:

```
session['name'] = user.name
```

Simply add the above code to the `if` block in the `login()` function. Also, be sure to update the `logout()` function as well:

```
@users_blueprint.route('/logout/')
@login_required
def logout():
    session.pop('logged_in', None)
    session.pop('user_id', None)
    session.pop('role', None)
    session.pop('name', None)
    flash('Goodbye!')
    return redirect(url_for('users.login'))
```

Next, update the `tasks()` function within `tasks/views.py` to pass in the `username` to the template:

```
@tasks_blueprint.route('/tasks/')
@login_required
def tasks():
    return render_template(
        'tasks.html',
        form=AddTaskForm(request.form),
        open_tasks=open_tasks(),
        closed_tasks=closed_tasks(),
        username=session['name']
    )
```

Finally, update the navbar within parent template in `templates/_base.html`:

```
<div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="/">FlaskTaskr</a>
        </div>
        <div class="collapse navbar-collapse">
            <ul class="nav navbar-nav">
                {% if not session.logged_in %}
                    <li><a href="{{ url_for('users.register') }}">Signup</a></li>
                {% else %}
                    <li><a href="{{ url_for('users.logout') }}">Signout</a></li>
                {% endif %}
            </ul>
            {% if session.logged_in %}
                <ul class="nav navbar-nav navbar-right">
                    <li><a>Welcome, {{username}}.</a></li>
                </ul>
            {% endif %}
        </div><!-- .nav-collapse -->
    </div>
</div>
```

So, here we test whether the `logged_in` key is in the session object - `{% if session.logged_in %}` - then we display the username like so: `<a>Welcome, {{username}}.`.

That's it. Easy, right? Run the tests again.

Display Update and Delete links

Since users can only modify tasks that they originally added, let's only display the 'Mark as Complete' and 'Delete' links for tasks that users are able to modify. Remember: Users with the role of `admin` can modify all posts.

Test

```
def test_users_cannot_see_task_modify_links_for_tasks_not_created_by_them(self):
    :
        self.register('Michael', 'michael@realpython.com', 'python', 'python')
        self.login('Michael', 'python')
        self.app.get('tasks/', follow_redirects=True)
        self.create_task()
        self.logout()
        self.register(
            'Fletcher', 'fletcher@realpython.com', 'python101', 'python101'
        )
        response = self.login('Fletcher', 'python101')
        self.app.get('tasks/', follow_redirects=True)
        self.assertNotIn(b'Mark as complete', response.data)
        self.assertNotIn(b'Delete', response.data)

def test_users_can_see_task_modify_links_for_tasks_created_by_them(self):
    self.register('Michael', 'michael@realpython.com', 'python', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    self.logout()
    self.register(
        'Fletcher', 'fletcher@realpython.com', 'python101', 'python101'
    )
    self.login('Fletcher', 'python101')
    self.app.get('tasks/', follow_redirects=True)
    response = self.create_task()
    self.assertIn(b'complete/2/', response.data)
    self.assertIn(b'complete/2/', response.data)

def test_admin_users_can_see_task_modify_links_for_all_tasks(self):
    self.register('Michael', 'michael@realpython.com', 'python', 'python')
    self.login('Michael', 'python')
    self.app.get('tasks/', follow_redirects=True)
    self.create_task()
    self.logout()
    self.create_admin_user()
    self.login('Superman', 'allpowerful')
    self.app.get('tasks/', follow_redirects=True)
    response = self.create_task()
    self.assertIn(b'complete/1/', response.data)
    self.assertIn(b'delete/1/', response.data)
    self.assertIn(b'complete/2/', response.data)
    self.assertIn(b'delete/2/', response.data)
```

Look closely at these tests before you run them. Will they both fail? Run them. You should only get one failure because the links already exist for all tasks. Essentially, the last two tests are to ensure that no regressions occur when we write the code to make

the first test pass.

Code

As for the code, we simply need to make a few changes in the *project/templates/tasks.html* template:

```
{% for task in open_tasks %}
<tr>
  <td>{{ task.name }}</td>
  <td>{{ task.due_date }}</td>
  <td>{{ task.posted_date }}</td>
  <td>{{ task.priority }}</td>
  <td>{{ task.poster.name }}</td>
  <td>
    {% if task.poster.name == session.name or session.role == "admin" %}
      <a href="{{ url_for('tasks.delete_entry', task_id = task.task_id) }}>Delete</a> -
      <a href="{{ url_for('tasks.complete', task_id = task.task_id) }}>Mark as Complete</a>
    {% else %}
      <span>N/A</span>
    {% endif %}
  </td>
</tr>
{% endfor %}
```

and

```
{% for task in closed_tasks %}
<tr>
  <td>{{ task.name }}</td>
  <td>{{ task.due_date }}</td>
  <td>{{ task.posted_date }}</td>
  <td>{{ task.priority }}</td>
  <td>{{ task.poster.name }}</td>
  <td>
    {% if task.poster.name == session.name or session.role == "admin" %}
      <a href="{{ url_for('tasks.delete_entry', task_id = task.task_id) }}>Delete</a>
    {% else %}
      <span>N/A</span>
    {% endif %}
  </td>
</tr>
{% endfor %}
```

Essentially, we're testing to see if the name of the poster matches the name of the user in the session and whether the user's role is `admin`. If either test evaluates to `True`, then we display the links to modify tasks.

Run the tests again to make sure they all pass.

That's it for the added features. What else would you like to see? Implement it utilizing test driven development.

Password Hashing

Right now our passwords are saved as plain text in the database, we need to securely [hash](#) them before they are added to the database to prevent them from being recovered by a hostile outsider. Keep in mind that unless you are a cryptographer with an advanced degree in computer science or mathematics you should never write your own cryptographic hasher. There's no need to reinvent the wheel. The problem of securely storing passwords has already been solved.

SEE ALSO: Want more information regarding securely storing passwords? The Open Web Application Security Project (OWASP), one of the application security industry's most trusted resource, provides a number of [recommendations](#) for secure password storage. Highly recommended.

Setup Flask-Bcrypt

Let's use a Flask extension called [Flask-Bcrypt](#):

```
$ pip install flask-bcrypt
$ pip freeze > requirements.txt
```

To set up the extension, simply import the class wrapper and pass the Flask app object into it:

```
# project/__init__.py

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt

app = Flask(__name__)
app.config.from_pyfile('_config.py')
bcrypt = Bcrypt(app)
db = SQLAlchemy(app)

from project.users.views import users_blueprint
from project.tasks.views import tasks_blueprint

# register our blueprints
app.register_blueprint(users_blueprint)
app.register_blueprint(tasks_blueprint)
```

Then update *users/views.py* so that password is hashed during registration:

```
if form.validate_on_submit():
    new_user = User(
        form.name.data,
        form.email.data,
        bcrypt.generate_password_hash(form.password.data)
    )
```

The `login()` function also needs to be updated:

```
@users_blueprint.route('/', methods=['GET', 'POST'])
def login():
    error = None
    form = LoginForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            user = User.query.filter_by(name=request.form['name']).first()
            if user is not None and bcrypt.check_password_hash(
                user.password, request.form['password']):
                session['logged_in'] = True
                session['user_id'] = user.id
                session['role'] = user.role
                session['name'] = user.name
                flash('Welcome!')
                return redirect(url_for('tasks.tasks'))
            else:
                error = 'Invalid username or password.'
    return render_template('login.html', form=form, error=error)
```

Don't forget to update the imports:

```
from project import db, bcrypt
```

Manual Test

Delete the database. Then update the `db_create.py` script:

```
# db_create.py

from project import db

# create the database and the db table
db.create_all()

# commit the changes
db.session.commit()
```

Fire up the server. Register a new user. Log in.

Unit Test

Run the test suite:

```
$ nosetests --with-coverage --cover-erase --cover-package=project
EEE....EEE.....
```

Name	Stmts	Miss	Cover
project.py	11	0	100%
project._config.py	7	0	100%
project.models.py	35	2	94%
project.tasks.py	0	0	100%
project.tasks.forms.py	9	0	100%
project.tasks.views.py	55	19	65%
project.users.py	0	0	100%
project.users.forms.py	11	0	100%
project.users.views.py	52	0	100%
TOTAL	180	21	88%

```
Ran 29 tests in 9.979s
```

```
FAILED (errors=9)
```

Take a look at the errors: `ValueError: Invalid salt`. Essentially, since we are manually hashing the password in the view, we need to do the same in every other place we create a password. Update the following helper methods within both of the test files:

```
def create_user(self, name, email, password):
    new_user = User(
        name=name,
        email=email,
        password=bcrypt.generate_password_hash(password)
    )
    db.session.add(new_user)
    db.session.commit()
```

and

```
def create_admin_user(self):
    new_user = User(
        name='Superman',
        email='admin@realpython.com',
        password=bcrypt.generate_password_hash('allpowerful'),
        role='admin'
    )
    db.session.add(new_user)
    db.session.commit()
```

```
def test_users_can_register(self):
    new_user = User("michael", "michael@mherman.org", bcrypt.generate_password_hash('michaelherman'))
    db.session.add(new_user)
    db.session.commit()
    test = db.session.query(User).all()
    for t in test:
        t.name
    assert t.name == "michael"
```

Make sure to update the imports:

```
from project import app, db, bcrypt
```

That's it. Run the tests again. They all should pass.

Custom Error Pages

Flask comes with a nice `errorhandler()` [function](#) for defining custom error pages, used for throwing [HTTP exceptions](#). Let's look at a quick example.

First, we need to write our tests...

Test

Let's add these to a new test file called `test_main.py`:

```
# project/test_main.py

import os
import unittest

from project import app, db
from project._config import basedir
from project.models import User

TEST_DB = 'test.db'

class MainTests(unittest.TestCase):

    #####
    #### setup and teardown ####
    #####
    # executed prior to each test
    def setUp(self):
        app.config['TESTING'] = True
        app.config['WTF_CSRF_ENABLED'] = False
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///\\' + \
            os.path.join(basedir, TEST_DB)
        self.app = app.test_client()
        db.create_all()

    # executed after each test
    def tearDown(self):
        db.session.remove()
        db.drop_all()
```

```
#####
##### helper methods #####
#####

def login(self, name, password):
    return self.app.post('/', data=dict(
        name=name, password=password), follow_redirects=True)

#####
##### tests #####
#####

def test_404_error(self):
    response = self.app.get('/this-route-does-not-exist/')
    self.assertEqual(response.status_code, 404)

def test_500_error(self):
    bad_user = User(
        name='Jeremy',
        email='jeremy@realpython.com',
        password='django'
    )
    db.session.add(bad_user)
    db.session.commit()
    response = self.login('Jeremy', 'django')
    self.assertEqual(response.status_code, 500)

if __name__ == "__main__":
    unittest.main()
```

Now when you run the tests, only the `test_500_error()` error fails: `ValueError: Invalid salt`. Remember: It's a good practice to not only check the status code, but some text on the rendered HTML as well. In the case of the 404 error, we have no idea what the end user sees on their end.

Let's check that. Fire up your server. Navigate to <http://localhost:5000/this-route-does-not-exist/>, and you should see the generic, black-and-white 404 Not Found page.

Is that really what we want the end user to see? Probably not. Let's handle that more gracefully. Think about what you'd like the user to see and then update the test:

```
def test_404_error(self):
    response = self.app.get('/this-route-does-not-exist/')
    self.assertEqual(response.status_code, 404)
    self.assertIn(b'Sorry. There\'s nothing here.', response.data)
```

Run the tests again: `FAILED (errors=1, failures=1)`.

So, what do you think the end user sees when the exception, `ValueError: Invalid salt`, is thrown? Let's assume the worst: They probably see that exact same ugly error.

Update the test to ensure that (a) the client does not see that error and (b) that the text we do want them to see is visible in the response:

```
def test_500_error(self):
    bad_user = User(
        name='Jeremy',
        email='jeremy@realpython.com',
        password='django'
    )
    db.session.add(bad_user)
    db.session.commit()
    response = self.login('Jeremy', 'django')
    self.assertEqual(response.status_code, 500)
    self.assertNotIn(b'ValueError: Invalid salt', response.data)
    self.assertIn(b'Something went terribly wrong.', response.data)
```

Run the tests: `FAILED (errors=1, failures=1)`. Nice.

Error Handler 404

View

The error handlers are set just like any other view. Update `project/__init__.py`:

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

This function catches the 404 error, and replaces the default template with `404.html` (which we will create next). Notice how we also have to specify the `status code`, `404`, we want thrown as well.

Make sure to import `render_template`.

Template

Create a new template in `project/templates` called `404.html`:

```
{% extends "_base.html" %}

{% block content %}

<h1>404</h1>
<p>Sorry. There's nothing here.</p>
<p><a href="{{url_for('users.login')}}">Go back home</a></p>

{% endblock %}
```

Test

`test_404_error` should now pass. Manually test it again as well: Navigate to <http://localhost:5000/this-route-does-not-exist/>. Nice.

Error Handler 500

View

```
update *poect/\\init\\_.py :
```

```
@app.errorhandler(500)
def internal_error(error):
    return render_template('500.html'), 500
```

Based on the last 404 `errorhandler()` can you describe what's happening here?

Template

Create a new template in `project/templates` called `500.html`:

```
{% extends "_base.html" %}

{% block content %}

<h1>500</h1>
<p>Something went terribly wrong. Fortunately we are working to fix it right now!
</p>
<p><a href="{{url_for('users.login')}}">Go back home</a></p>

{% endblock %}
```

Test

If you run the tests, you'll see we still have an that invalid salt error. Update the the

```
test_500_error :
```

```
def test_500_error(self):
    bad_user = User(
        name='Jeremy',
        email='jeremy@realpython.com',
        password='django'
    )
    db.session.add(bad_user)
    db.session.commit()
    self.assertRaises(ValueError, self.login, 'Jeremy', 'django')
    try:
        response = self.login('Jeremy', 'django')
        self.assertEquals(response.status_code, 500)
    except ValueError:
        pass
```

Nice! All tests passing. No un-handled errors.

Now, let's manually test this to see what's actually happening for the user. Temporarily update the part of the `register()` function, in `users/views.py`, where the new user is created.

Change:

```
new_user = User(
    form.name.data,
    form.email.data,
    bcrypt.generate_password_hash(form.password.data)
)
```

To:

```
new_user = User(
    form.name.data,
    form.email.data,
    form.password.data
)
```

Now when a new user is registered, the password will not be hashed. Test this out. Fire up the server. Register a new user, and then try to log in. You should see the Flask debug page come up, with the `ValueError: Invalid salt`. Why is this page populating? Simple: `debug mode` is on in the `run.py` file: `debug=True`.

Since we never want this on in a production environment, let's manually change it to `False` to see the actual error that the end user will see. Fire up the server again, register another new user, and then try to log in. Now you should see our custom 500 error page.

Make sure to add `bcrypt` back to the view, `users/views.py`:

```
new_user = User(  
    form.name.data,  
    form.email.data,  
    bcrypt.generate_password_hash(form.password.data)  
)
```

Also, notice that when we run the tests again the test still doesn't pass. Why? Part of the problem has to do, again, with debug mode set to `True`. Let's fix that since tests should *always* run with debug mode off.

Update Debug Mode

Add `DEBUG = True` to the `_config.py` file and then update `run.py`:

```
# run.py  
  
from project import app  
app.run()
```

Fire up the server. It should still run in debug mode. Now let's update `setup()` for all test files:

```
def setUp(self):
    app.config['TESTING'] = True
    app.config['WTF_CSRF_ENABLED'] = False
    app.config['DEBUG'] = False
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///{}' + \
        os.path.join(basedir, TEST_DB)
    self.app = app.test_client()
    db.create_all()

    self.assertEquals(app.debug, False)
```

Here, we explicitly turn off debug mode - `app.config['DEBUG'] = False` - and then also run a test to ensure that for each test ran, it's set to `False` -

`self.assertEquals(app.debug, False)` . Run your tests again. The `test_500_error` test still doesn't pass. Go ahead and remove it for now. We'll discuss why later and how to set this test up right. For now, let's move on to logging all errors in our app.

Let's face it, your code will never be 100% error free, and you will never be able to write tests to cover *everything* that could potentially happen. Thus, it's vital that you set up a means of capturing all errors so that you can spot trends, setup additional error handlers, and, of course, fix errors that occur.

It's very easy to setup logging at the server level or within the Flask application itself. In fact, if your app is running in production, by default errors are added to the [logger](#). You can then grab those errors and log them to a file as well as have the most critical errors sent to you via email. There are also numerous third party libraries that can manage the logging of errors on your behalf.

At the very least, you should set up a means of emailing critical errors since those should be addressed immediately. Please see the official [documentation](#) for setting this up. It's easy to setup, so we won't cover it here. Instead, let's build our own custom solution to log errors to a file.

Here's a basic logger that uses the [logging](#) library. Update the error handler views in `project/__init__.py` :

```
@app.errorhandler(404)
def page_not_found(error):
    if app.debug is not True:
        now = datetime.datetime.now()
        r = request.url
        with open('error.log', 'a') as f:
            current_timestamp = now.strftime("%d-%m-%Y %H:%M:%S")
            f.write("\n404 error at {}: {}".format(current_timestamp, r))
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    if app.debug is not True:
        now = datetime.datetime.now()
        r = request.url
        with open('error.log', 'a') as f:
            current_timestamp = now.strftime("%d-%m-%Y %H:%M:%S")
            f.write("\n500 error at {}: {}".format(current_timestamp, r))
    return render_template('500.html'), 500
```

Be sure to update the imports as well:

```
import datetime
from flask import Flask, render_template, request
from flask.ext.sqlalchemy import SQLAlchemy
from flask.ext.bcrypt import Bcrypt
```

The code is simple: When an error occurs we add the timestamp and the request to an error log, `error.log`. Notice how these errors are only logged when debug mode is off. Why do you think we would want that? Well, as you know, when the debug mode is on, errors are caught by the Flask debugger and then handled gracefully by displaying a nice formatted page with info on how to correct the issue. Since this is caught by the debugger, it will not throw the right errors. Further, since debug mode will always be off in production, these errors will be caught by the custom error logger. Make sense?

Save the code. Turn debug mode off. Fire up the server, then navigate to http://localhost:5000/does_not_exist. You should see the `error.log` file with the following error logged:

```
404 error at 17-10-2016 19:51:08: http://localhost:5000/does_not_exist
```

Let's try a 500 error. Again, temporarily update the part of the `register()` function, in `users/views.py`:

Change:

```
new_user = User(  
    form.name.data,  
    form.email.data,  
    bcrypt.generate_password_hash(form.password.data)  
)
```

To:

```
new_user = User(  
    form.name.data,  
    form.email.data,  
    form.password.data  
)
```

Kill the server. Fire it back up. Register a new user, and then try to login. You should see the 500 error page the following error in the error log:

```
500 error at 17-10-2016 19:51:10: http://localhost:5000/
```

That's it. Make sure to add `bcrypt` back to `users/views.py` and turn debug mode back on.

Flask: FlaskTaskr, Part 7 - Deployment

Where are we at with the app?

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	Yes
Testing	Yes
Styling	Yes
Test Coverage	Yes
Nose Testing Framework	Yes
Permissions	Yes
Blueprints	Yes
New Features	Yes
Password Hashing	Yes
Custom Error Pages	Yes
Error Logging	Yes
Deployment Options	No
Automated Deployments	No
Building a REST API	No
Continuous Integration and Delivery	No

Deployment

As far as deployment options go, [PythonAnywhere](#) and [Heroku](#) are great. We'll use PythonAnywhere throughout the web2py sections, so let's use Heroku with this app.

SEE ALSO: Check out the official Heroku [docs](#) for deploying an app if you need additional help.

Setup

Deploying an app to Heroku is ridiculously easy:

1. Sign up for [Heroku](#).
2. Log in and download the [Heroku Toolbelt](#).
3. Once installed, open your terminal and run the following command: `heroku login`. You'll need to enter your email and password you used when you signed up for Heroku.
4. Navigate to your project and activate the virtual environment.
5. Install [gunicorn](#): `pip install gunicorn`.
6. Heroku recognizes the dependencies needed through a *requirements.txt* file. Make sure to update your file: `pip freeze > requirements.txt`.
7. Create a [Procfile](#). Open up a text editor and save the following text in it: `web: python run.py`. Then save the file in your application's root or main directory as *Procfile* (no extension). The word "web" indicates to Heroku that the application will be attached to HTTP once deployed to Heroku.

On your local machine, the application runs on port 5000 by default. On Heroku, the application **must** run on a random port specified by Heroku. We will identify this port number by reading the environment variable 'PORT' and passing it to `app.run`:

```
# run.py

import os
from project import app

port = int(os.environ.get('PORT', 5000))
app.run(host='0.0.0.0', port=port)
```

Set debug to `False` within the config file. Why? Again, debug mode provides a handy debugger for when errors occur, which is great during development, but you never want end users to see this. It's a security vulnerability, as it is possible to execute commands through the debugger.

Deploy

When you PUSH to Heroku, you have to update your local Git repository. Commit your updated code.

Create your app on Heroku:

```
$ heroku create
```

Deploy your code to Heroku:

```
$ git push heroku master
```

Add a PostgreSQL database:

```
$ heroku addons:create heroku-postgresql:dev
```

Check to make sure your app is running:

```
$ heroku ps
```

View the app in your browser:

```
$ heroku open
```

If you see errors, open the Heroku log to view all errors and output:

```
$ heroku logs
```

Finally, run your tests on Heroku:

```
$ heroku run nosetests
Running `nosetests` attached to terminal... up, run.9405
.
.
.
-----
Ran 30 tests in 24.469s

OK
```

All should pass.

That's it. Make sure that you also PUSH your local repository to Github.

SEE ALSO: You can see my app at <http://flasktaskr.herokuapp.com>. Cheers!

Automated Deployments

In the last lesson we manually uploaded our code to Heroku. When you only need to deploy code to Heroku and GitHub, you can get away with doing this manually. However, if you are working with multiple servers where a number of developers are sending code each day, you will want to automate this process. We can use [Fabric](#) for such automation.

WARNING: As of writing (October 18, 2016), Fabric only works with Python 2.x.

Please review the [Development roadmap](#) for more information.

Setup

As always, start by installing the dependency with Pip:

PETE: Can you update this to the latest version of fabric

```
$ pip install fabric==1.12.0
$ pip freeze > requirements.txt
```

Fabric is controlled by a file called a fabfile, *fabfile.py*. You define all of the actions (or commands) that Fabric takes in this file. Create the file within your app's root directory.

The file itself takes a number of commands. Here's a brief list of some of the most common commands:

- `run` runs a command on a remote server
- `local` runs a local command
- `put` uploads a local file to the remote server
- `cd` changes the directory on the server side
- `get` downloads a file from the remote server
- `prompt` prompts a user with text and returns the user input

Preparing

Add the following code to *fabfile.py*:

```

def test():
    local("nosetests -v")

def commit():
    message = raw_input("Enter a git commit message: ")
    local("git add . && git commit -am '{}'".format(message))

def push():
    local("git push origin master")

def prepare():
    test()
    commit()
    push()

```

Essentially, we imported the `local` method from Fabric, then ran the basic shell commands for testing and PUSHing to GitHub as you've seen before.

Test this out:

```
$ fab prepare
```

If all goes well, this should run the tests, commit the code to your local repo, and then deploy to GitHub.

Testing

What happens if your tests fail? Wouldn't you want to abort the process? Probably.

Update your `test()` function to:

```

def test():
    with settings(warn_only=True):
        result = local("nosetests -v", capture=True)
    if result.failed and not confirm("Tests failed. Continue?"):
        abort("Aborted at user request.")

```

Also update the imports:

```

from fabric.api import local, settings, abort
from fabric.contrib.console import confirm

```

Here, if a test fails, then the user is asked to confirm whether or not the script should continue running.

Deploying

Finally, let's deploy to Heroku as well:

```
def pull():
    local("git pull origin master")

def heroku():
    local("git push heroku master")

def heroku_test():
    local("heroku run nosetests -v")

def deploy():
    pull()
    test()
    commit()
    heroku()
    heroku_test()
```

Now when you run the `deploy()` function, you PULL the latest code from GitHub, test the code, commit it to your local repo, PUSH to Heroku, and then test on Heroku. The commands should all look familiar.

The last thing we need to add is a quick rollback.

```
def rollback():
    local("heroku rollback")
```

Updated file:

```
from fabric.api import local, settings, abort
from fabric.contrib.console import confirm

# prep

def test():
    with settings(warn_only=True):
        result = local("nosetests -v", capture=True)
    if result.failed and not confirm("Tests failed. Continue?"):
        abort("Aborted at user request.")
```

```
def commit():
    message = raw_input("Enter a git commit message: ")
    local("git add . && git commit -am '{}'".format(message))

def push():
    local("git push origin master")

def prepare():
    test()
    commit()
    push()

# deploy

def pull():
    local("git pull origin master")

def heroku():
    local("git push heroku master")

def heroku_test():
    local("heroku run nosetests -v")

def deploy():
    # pull()
    test()
    # commit()
    heroku()
    heroku_test()

# rollback

def rollback():
    local("heroku rollback")
```

If you do run into an error on Heroku, you want to immediately load a prior commit to get it working. You can do this quickly now by running the command:

```
$ fab rollback
```

Keep in mind that this is just a temporary fix. After you rollback, make sure to fix the issue locally and then PUSH the updated code to Heroku.

Flask: FlaskTaskr, Part 8 - RESTful API

Where are we at with the app?

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	Yes
Testing	Yes
Styling	Yes
Test Coverage	Yes
Nose Testing Framework	Yes
Permissions	Yes
Blueprints	Yes
New Features	Yes
Password Hashing	Yes
Custom Error Pages	Yes
Error Logging	Yes
Deployment Options	Yes
Automated Deployments	Yes
Building a REST API	No
Continuous Integration and Delivery	No

Building a RESTful API

Finally, let's take a quick look at how to design a RESTful API in Flask. We'll look more at APIs in an upcoming chapter. For now we'll just define the basics, then build the actual API.

What's an API?

Put simply, an API is collection of functions that other programs can use to access or manipulate data from a determine database. Each function has an associated endpoint (also called a resource). One can make changes to a resource via the HTTP methods/verbs:

1. GET - view a resource
2. POST - create a new resource
3. PUT - update a resource
4. DELETE - delete a resource

WARNING: When you set up a RESTful API, you expose much of your app's internal system to the rest of the world (or even other areas within your own organization). Keep this in mind. Only allow users or programs access to a limited set of data, using only the needed HTTP methods. Never expose more than necessary.

Basic RESTful Design Practices

URLs (endpoints) are used for identifying a specific resource, while the HTTP methods define the actions one can take against those resources. Each resource should only have, at most, two URLs. The first is for a collection, while the second is for a specific element in that collection.

For example, in the FlaskTaskr app, the endpoint `/tasks/` could be for the collection, while `/tasks/<id>/` could be for a specific element (e.g., a single task) from the collection.

NOTE: If you're using an ORM, like SQLAlchemy, more often than not, resources are generally your models.

	GET	POST	PUT	DELETE
/tasks/	View all tasks	Add task	Update all tasks	Delete all tasks
/tasks/<id>/	View single task	N/A	Update single task	Delete single task

NOTE: In our API, we are only going to be working with the GET request since it is read-only. In other words, we do not want external users to add or manipulate data.

Workflow

Workflow for creating an API via Flask:

1. Source the data
2. Set up a persistence layer
3. Install Flask
4. Setup/Install Flask-SQLAlchemy
5. Create URLs/Endpoints
6. Add Query Strings/Parameters to your URLs (optional)
7. Test, Test, Test

NOTE: This workflow is just for the backend. We'll be responding with JSON based on a user request - but that's it. The frontend of the app could also utilize this data as well. For example, you could use jQuery or a front-end library or framework such as React, Ember, or Angular to consume the data from the API and display it to the end user.

We already have the first four steps done, so let's start with creating our actual endpoints. Also, as of right now, we're not going to have the ability to add specific operations with query strings.

With that, let's set up our endpoints...

First Endpoint

From the design above, the two URLs we want our app to support are `/tasks/` and `/tasks/<id>/`. Let's start with the former.

Test

First, let's add a new test file to test our API code. Name the file `test_api.py`.

Pull up the `test_api.py` file from the inside `assests` folder within the [exercises repo](#) and cut and paste it into you project.

Have a look at the code.

What's going on?

1. Like our previous tests, we first set up a test client along with the `setup()` and `tearDown()` methods.
2. The helper method, `add_tasks()`, is used to add dummy tasks to the test database.
3. Then when we run the test, we add the tasks to the DB, hit the endpoint, then test that the appropriate response is returned.

What should happen when you run the tests? It will fail, of course.

Code

Start by creating a new Blueprint called "api". You just need two files - `__init__.py` and `views.py`.

Pull up the `api/views.py` file from the inside `assests` folder within the [exercises repo](#) and cut and paste it into you project.

What's going on?

1. We map the URL `'/api/v1/tasks/'` to the `api_tasks()` function so once that URL is requested via a GET request, we query the database to grab the first 10 records from the `tasks` table.
2. Next we create a dictionary out of each returned record from the database.
3. Finally, since our API supports JSON, we pass in the dictionary to the `jsonify()` function to render a JSON response back to the browser.

SEE ALSO: Take a minute to read about the `jsonify()` function from the official [Flask documentation](#). This is a powerful function, used to simplify and beautify your code. It not only takes care of serialization, but it also validates the data itself and adds the appropriate status code and header. Without it, your response object would need to look something like this: `return json.dumps(data), 200, { "Content-Type" : "application/json"}`

Make sure to register the Blueprint in `project/__init__.py`:

```
from project.users.views import users_blueprint
from project.tasks.views import tasks_blueprint
from project.api.views import api_blueprint

# register our blueprints
app.register_blueprint(users_blueprint)
app.register_blueprint(tasks_blueprint)
app.register_blueprint(api_blueprint)
```

Turn debug mode back on. Navigate to <http://localhost:5000/api/v1/tasks/> to view the returned JSON after you've fired up the server.

Test Again

Does the new test pass?

```
$ nosetests tests/test_api.py
.
-----
Ran 1 test in 0.050s

OK
```

Of course!

NOTE: Notice how we isolated the tests to just test *test_api.py*. This speeds up development as we do not have to run the entire test suite each time. That said, be sure to run all the tests after you finish implementing a feature to make sure there are no regressions.

Second Endpoint

Now, let's add the next endpoint.

Test

```
def test_resource_endpoint_returns_correct_data(self):
    self.add_tasks()
    response = self.app.get('api/v1/tasks/2', follow_redirects=True)
    self.assertEquals(response.status_code, 200)
    self.assertEquals(response.mimetype, 'application/json')
    self.assertIn(b'Purchase Real Python', response.data)
    self.assertNotIn(b'Run around in circles', response.data)
```

This is very similar to our first test.

Write the Code

```
@api_blueprint.route('/api/v1/tasks/<int:task_id>')
def task(task_id):
    result = db.session.query(Task).filter_by(task_id=task_id).first()
    json_result = {
        'task_id': result.task_id,
        'task name': result.name,
        'due date': str(result.due_date),
        'priority': result.priority,
        'posted date': str(result.posted_date),
        'status': result.status,
        'user id': result.user_id
    }
    return jsonify(items=json_result)
```

What's going on?

This is very similar to our last endpoint. The difference is that we are using a dynamic route to grab a query and render a specific `task_id`. Manually test this out. Navigate to <http://localhost:5000/api/v1/tasks/1>. You should see a single task.

Test Again

```
$ nosetests tests/test_api.py
.
-----
Ran 2 tests in 0.068s
OK
```

Alright, so all looks good, right?

Wrong. What happens if the URL you hit is for a `task_id` that does not exist? Try it. Navigate to <http://localhost:5000/api/v1/tasks/209> in your browser. You should see a 500 error. It's okay for the element not to exist, but we need to return a user-friendly error. Let's update the code.

Updated Endpoints

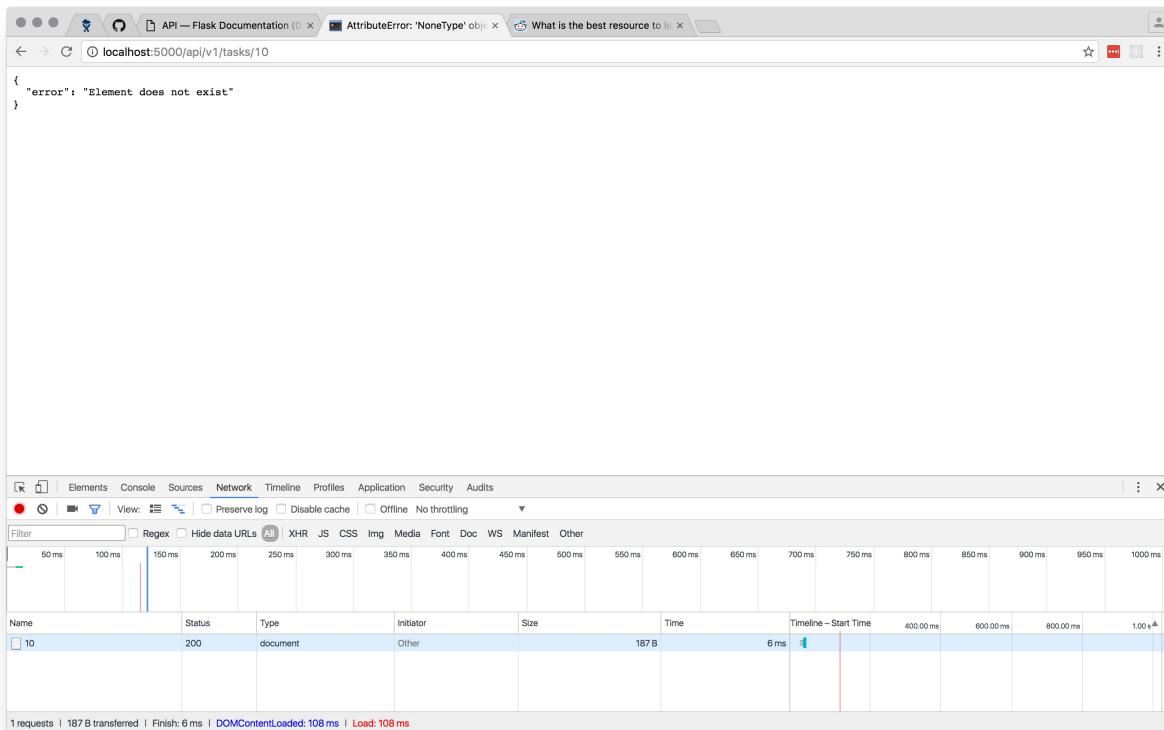
Test

```
def test_invalid_resource_endpoint_returns_error(self):
    self.add_tasks()
    response = self.app.get('api/v1/tasks/209', follow_redirects=True)
    self.assertEqual(response.status_code, 404)
    self.assertEqual(response.mimetype, 'application/json')
    self.assertIn(b'Element does not exist', response.data)
```

Write the Code

```
@api_blueprint.route('/api/v1/tasks/<int:task_id>')
def task(task_id):
    result = db.session.query(Task).filter_by(task_id=task_id).first()
    if result:
        json_result = {
            'task_id': result.task_id,
            'task name': result.name,
            'due date': str(result.due_date),
            'priority': result.priority,
            'posted date': str(result.posted_date),
            'status': result.status,
            'user id': result.user_id
        }
        return jsonify(items=json_result)
    else:
        result = {"error": "Element does not exist"}
        return jsonify(result)
```

Before you test it out, open the Network tab within Chrome Developer Tools to confirm the status code.



Now test the endpoint <http://localhost:5000/api/v1/tasks/209>. We get the appropriate JSON response and content type, but we are seeing a 200 status code. Why? Well, since JSON is returned, Flask believes that this is a *good* endpoint. We really need to return a 404 status code in order to meet REST conventions.

Update the code again:

```
@api_blueprint.route('/api/v1/tasks/<int:task_id>')
def task(task_id):
    result = db.session.query(Task).filter_by(task_id=task_id).first()
    if result:
        result = {
            'task_id': result.task_id,
            'task name': result.name,
            'due date': str(result.due_date),
            'priority': result.priority,
            'posted date': str(result.posted_date),
            'status': result.status,
            'user id': result.user_id
        }
        code = 200
    else:
        result = {"error": "Element does not exist"}
        code = 404
    return make_response(jsonify(result), code)
```

Did you notice the new method `make_response()`? Read more about it [here](#). Be sure to import it as well.

Test Again

And manually test it again. You should see a 200 status code for a `task_id` that exists, as well as the appropriate JSON data, and a 404 status code for a `task_id` that does not exist. Good.

Now run your tests. They should all pass.

if not, did you remember to import the `make_response` method?

```
$ nosetests tests/test_api.py
...
-----
Ran 3 tests in 0.100s
OK
```

Let's also test the entire test suite to ensure we didn't break anything anywhere else in the codebase:

```
$ nosetests --with-coverage --cover-erase --cover-package=project
...
-----
```

Name	Stmts	Miss	Cover
project.py	31	8	74%
project._config.py	8	0	100%
project.api.py	0	0	100%
project.api.views.py	31	8	74%
project.models.py	35	2	94%
project.tasks.py	0	0	100%
project.tasks.forms.py	9	0	100%
project.tasks.views.py	55	0	100%
project.users.py	0	0	100%
project.users.forms.py	11	0	100%
project.users.views.py	52	0	100%
TOTAL	232	18	92%

```
-----
Ran 33 tests in 20.022s
OK
```

Perfect.

Deploy

Now deploy to Heroku!

```
$ git push heroku master
```

Homework

- Now that you've seen how to create a REST API from scratch, check out the [Flask-RESTful](#) extension that simplifies the process. Want a challenge? See if you can set up this extension, then add the ability for all users to POST new tasks. Once complete, give logged in users the ability to PUT (update) and DELETE individual elements within the collection - e.g., individual tasks.

Interlude: Flask Boilerplate Template and Workflow

You should now understand many of the underlying basics of creating an app in Flask. Let's look at a pre-configured Flask [boilerplate template](#) that has many of the bells and whistles installed so that you can get started creating an app right away. We'll also detail a workflow that you can use throughout the development process to help you stay organized and ensure that your Flask instance will scale right along with you.

Setup

First, navigate to a directory outside of your "realpython" directory. The "desktop" or "documents" directories are good choices since they are easy to access. Then [clone](#) the boilerplate template from Github:

```
$ git clone https://github.com/realpython/flask-skeleton.git
$ cd flask-skeleton
```

This creates a new folder called "flask-skeleton":

```
└── LICENSE
└── README.md
└── manage.py
└── project
    ├── __init__.py
    └── client
        ├── static
        │   ├── main.css
        │   └── main.js
        └── templates
            ├── _base.html
            ├── errors
            │   ├── 401.html
            │   ├── 403.html
            │   ├── 404.html
            │   └── 500.html
            ├── footer.html
            ├── header.html
            ├── main
            │   ├── about.html
            │   └── home.html
            └── user
                ├── login.html
                ├── members.html
                └── register.html
    └── server
        ├── __init__.py
        ├── config.py
        └── dev.sqlite
    └── main
        ├── __init__.py
        └── views.py
    └── models.py
    └── user
        ├── __init__.py
        ├── forms.py
        └── views.py
    └── tests
        ├── __init__.py
        ├── base.py
        ├── helpers.py
        ├── test_config.py
        ├── test_main.py
        └── test_user.py
└── requirements.txt
```

Activate a virtual environment, and then install the various libraries and dependencies:

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

You can also view the dependencies by running the command `pip freeze`:

```
alembic==0.8.9
bcrypt==3.1.1
blinker==1.4
cffi==1.9.1
click==6.6
coverage==4.2
dominate==2.3.1
Flask==0.11.1
Flask-Bcrypt==0.7.1
Flask-Bootstrap==3.3.7.0
Flask-DebugToolbar==0.10.0
Flask-Login==0.4.0
Flask-Migrate==2.0.2
Flask-Script==2.0.5
Flask-SQLAlchemy==2.1
Flask-Testing==0.6.1
Flask-WTF==0.13.1
itsdangerous==0.24
Jinja2==2.8
Mako==1.0.6
MarkupSafe==0.23
pycparser==2.17
python-editor==1.0.3
six==1.10.0
SQLAlchemy==1.1.4
visitor==0.1.3
Werkzeug==0.11.11
WTForms==2.1
```

Development workflow

Now that you have your skeleton app up, it's time to start developing locally.

1. Edit your application locally
2. Run and test locally
3. Push to Github
4. Push to Heroku
5. Test live application
6. Rinse and repeat

NOTE: Be sure to review the README to learn how to set up the database and run the app.

That's it. You now have a skeleton app to work with to build your own applications.
Cheers!

Flask: FlaskTaskr, Part 9: Continuous Integration and Delivery

We've talked about deployment a number of times already, but let's take it a step further by bridging the gap between development, testing, and deployment via Continuous Integration (CI) and Delivery (CD).

CI is a practice used by developers to help ensure that they are not introducing bugs into new code or causing old code to break. In practice, members of a development team integrate their code frequently, using a version control system like Git, which is then validated through the use of automated tests. Often times, such integrations happen many times a day. Plus, once the code is ready to be deployed (after many integrations), CD is used to reduce deployment times as well as errors by automating the process.

In many cases, a development workflow known as the [Feature Branch Workflow](#) is utilized, where a group of developers (or a single developer) work on a separate feature. Again, code is integrated numerous times daily and tests atomically run. Then once the feature is complete, a pull request is opened against the Master branch, which triggers another round of automated testing. If the tests pass, the Feature branch is merged into Master and then auto-deployed.

If you're new to Git, Github, or this workflow, check out the *Git Branching at a Glance* chapter in the third Real Python course as well as this excellent [tutorial](#). We will be using this workflow in this chapter.

Workflow

The workflow that we will use is fairly simple:

1. Create a new branch locally
2. Develop locally
3. Commit locally, then push to Github
4. Run the automated tests
5. Repeat until the feature is complete
6. Open a pull request against the Master branch
7. Run the automated tests
8. If the tests pass, deploy to Heroku

Before jumping in, let's quickly look at a few CI tools used to facilitate this process.

Continuous Integration Tools

There are a number of open source projects, like [Jenkins](#), [Buildbot](#), and [Strider](#), as well as hosted services that offer CI. In most cases they offer CD as well.

Most of the hosted services are either free for open source projects or have a free tier plan where you can run automated tests against a limited number of open Github repositories. There are a plethora of services out there - such as [Travis CI](#), [CircleCI](#), [Codeship](#), [Shippable](#), [Drone](#), [Snap](#), to name a few.

Bottom line: The CI projects are generally more powerful than the hosted services since you have full control. However, they take longer to set up and you have to continue to maintain them. The hosted services, on the other hand, are super easy to set up - most integrate with Github with a simple click of a button. Plus, you don't have to worry about whether or not your CI is setup correctly. It's not a good situation when you have to run tests against your CI tool to ensure it's running the tests against your app correctly.

For this project, we'll use [Travis CI](#) since it's battle tested, easy to integrate with Github, and free for open source projects.

Travis CI Setup

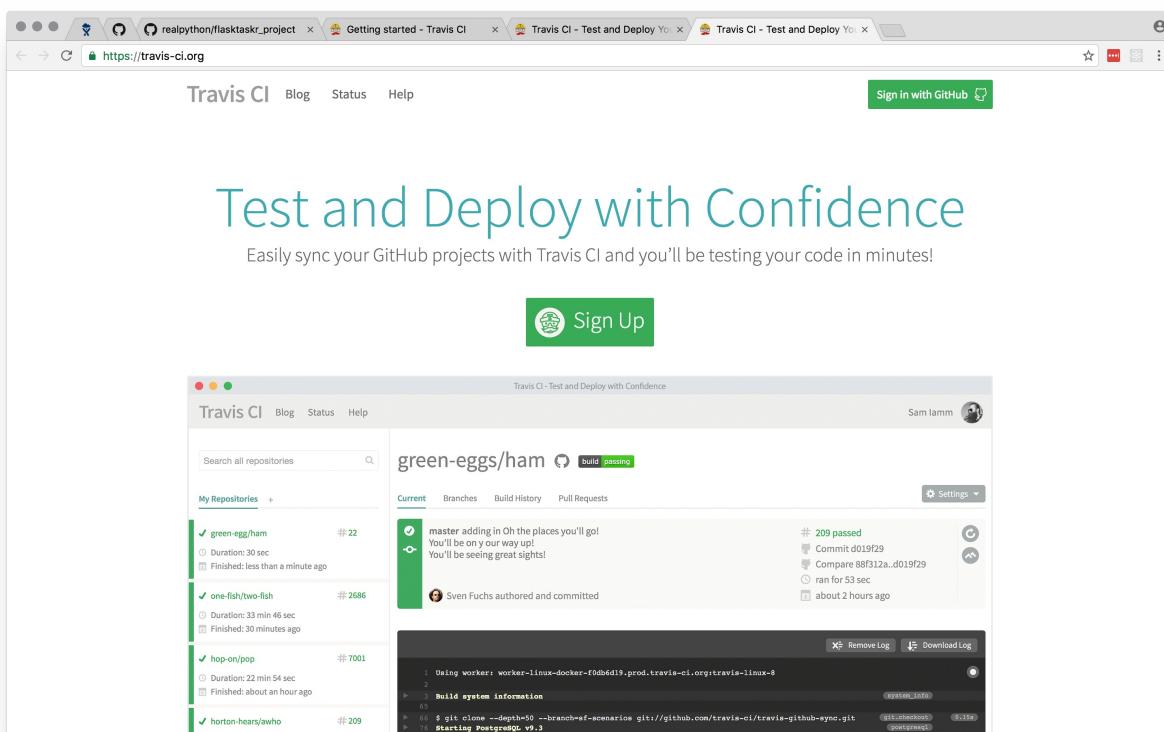
At this point you, you need to have accounts set up on Github and Heroku, a local Git repository, and a repository set up on Github for this project.

NOTE: The repository for this is outside the normal exercise repo. You can find it here - https://github.com/realpython/flasktaskr_project.

Quickly read over the Travis-CI [Getting Started](#) guide so you have a basic understanding of the process before following the steps in this tutorial.

Step 1: Sign up

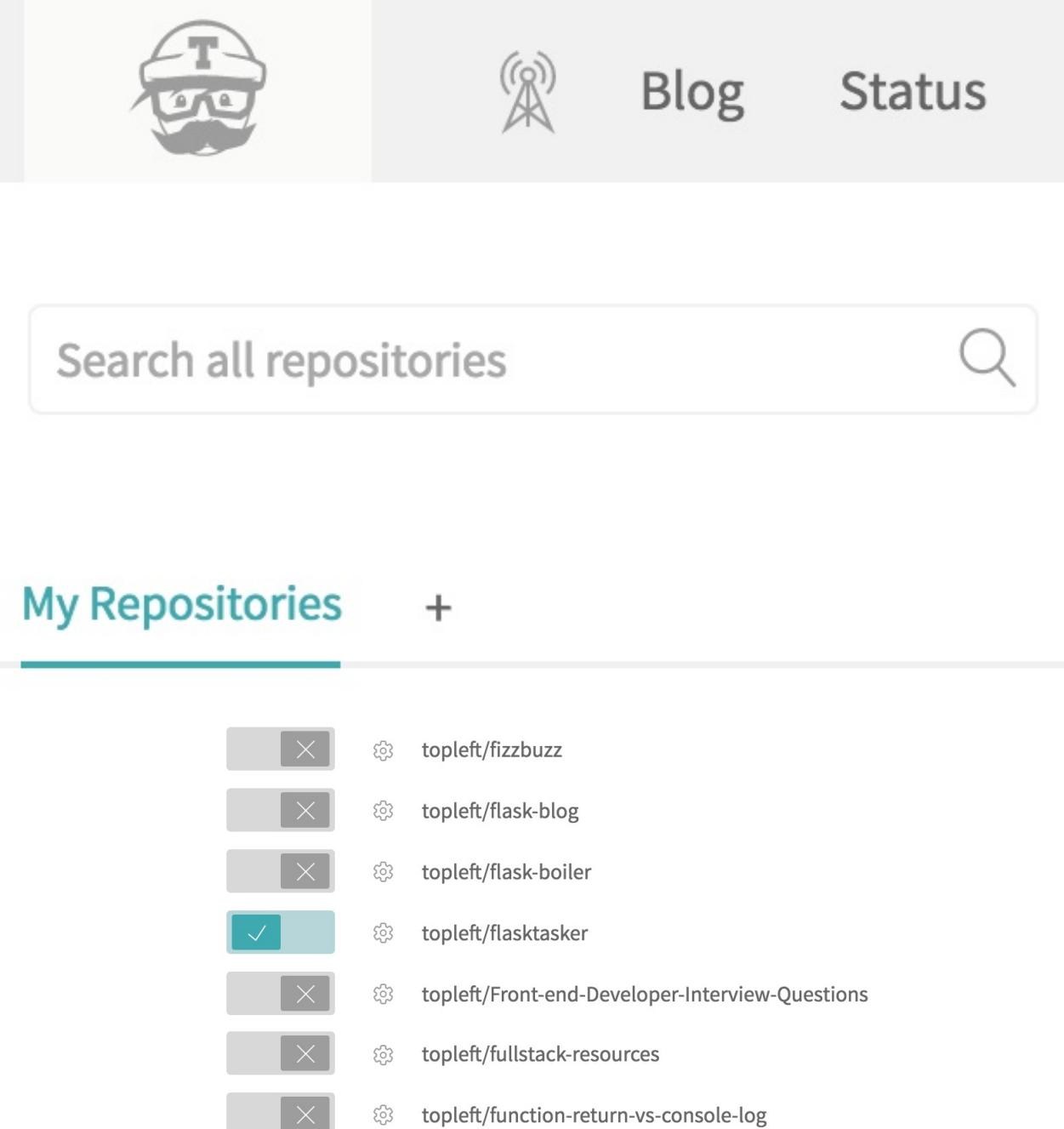
Navigate to <https://travis-ci.org/>, and then click "Sign in with GitHub".



You will be asked if you want to authorize Travis CI to access your account. Grant them access.

SEE ALSO: You can read more about the permissions asked for [here](#) when you grant Travis CI to access your Github account.

Once logged in, navigate to your profile page. There is a 'plus' button at the top left corner of the page. Click on it. You should be able to see all repositories that you have admin access to. Find the repository associated with this project and flip the switch to on to enable CI on it.



Search all repositories

My Repositories +

Repository	Status
topleft/fizzbuzz	disabled
topleft/flask-blog	disabled
topleft/flask-boiler	disabled
topleft/flasktasker	enabled
topleft/Front-end-Developer-Interview-Questions	disabled
topleft/fullstack-resources	disabled
topleft/function-return-vs-console-log	disabled

A Git Hook has been added to the repository so that Travis is triggered to run tests when new code is pushed to the repository.

Step 2: Add a Config File

Travis needs to know some basics about your project in order to run the tests correctly.

```
# specify language
language:
  - python

# specify python version(s)
python:
  - "3.6"
  - "2.7"

# install dependencies
install:
  - pip install -r requirements.txt

# run tests
script:
  - nosetests
```

Add this to a file called `.travis.yml` within the main directory. So we specified the language, the versions of Python we want to use to test against, and then the commands to install the project requirements and run the tests.

NOTE: This must be a valid [YAML](#) file. Copy and paste the contents of the file to the form on [Travis WebLint](#) to ensure that it is in proper YAML format.

Step 3: Trigger a New Build

Commit your code locally, and then push to Github. This will add a new build into the Travis CI queue. Wait for a worker to open up before the tests run. This can take a few minutes.

After the tests run - and pass - it's common to place the status within the `README` file on Github so that visitors can immediately see the build status. Read more about it [here](#). You can simply grab the markdown and add it to a `README.md` file.

Step 4: Travis Command Line Client

We need to install the [Travis Command Line Client](#). Unfortunately, you need to have Ruby installed first. Check out the [Travis CI documentation](#) for more info.

Running Windows?:

Check out [RubyInstaller](#).

Running OS X or Linux?:

Install via Homebrew:

Homebrew is a package manager for macOS and it is super useful for all kinds of installations and updates. If you don't already have homebrew installed, follow the instructions on their [homepage](#).

```
$ brew update  
$ brew install ruby
```

Once you have Ruby installed, run `gem install travis` to install the Travis Command Line Client. You may have to run `sudo gem install travis` to get around some permissions settings.

Now, within your project directory, run the following command:

```
$ travis setup heroku
```

Answer 'yes' to all the prompts.

```
18 gems installed  
→ local-flask-taskr git:(master) travis setup heroku  
Shell completion not installed. Would you like to install it now? lyl y  
Detected repository as topleft/flasktasker, is this correct? lyesl yes  
Deploy only from topleft/flasktasker? lyesl yes  
Encrypt API key? lyesl yes  
→ local-flask-taskr git:(master) x
```

This command automatically adds the necessary configuration for deploying to Heroku to the `deploy` section in the `.travis.yml` file, which should look something like this:

```
deploy:  
  provider: heroku  
  api_key:  
    secure: Dxdd3y/i7oTTsc5IHUebG/xVJIrGbzb1CCHm9KwE56  
  app: flasktaskr_project  
  on:  
    repo: realpython/flasktaskr_project
```

Commit your changes, then push to Github. The build should pass on Travis CI and then automatically push the new code to Heroku. Make sure your app is still running on Heroku after the push is complete.

SEE ALSO: For more information on setting up the Heroku configuration, please see the [official Travis CI documentation](#).

Intermission

Let's look what we've accomplished thus far in terms of CI and CD:

- Travis CI triggers the automated tests after we push code to the Master branch on Github.
- Then, If the tests pass, the code is deployed to Heroku.

With everything set up, let's jump back to the workflow to see what changes we need to make.

Workflow

Remember: The end goal is-

1. Create a new branch locally
2. Develop locally
3. Commit locally, then push to Github
4. Run the automated tests
5. Repeat until the feature is complete
6. Open a pull request against the Master branch
7. Run the automated tests
8. If the tests pass, deploy to Heroku

What changes need to be made based on this workflow vs. what we've developed thus far? Well, let's start with implementing the Feature Branch Workflow.

Feature Branch Workflow

The goal of this workflow is simple: since we'll be pushing code changes to the Feature branch rather than Master, deployments should only be triggered on the Master branch. In other words, we want Travis CI to run the automated tests each time we push to the Feature branch, but deployments are only triggered after we merge the Feature branch into Master. That way we can continue to develop on the Feature branch, frequently integrate changes, run tests, and then deploy only when we are finished with the feature and are ready to release it into the wild.

Update the Config File

So, we can specify the branch to deploy with the `on` option:

```
# specify language
language:
  - python

# specify python version(s)
python:
  - "3.6"
  - "2.7"

# install dependencies
install:
  - pip install -r requirements.txt

# run tests
script:
  - nosetests

# deploy!
deploy:
  provider: heroku
  api_key:
    secure: Kq95wr8v6rGlspuTt3Y8NdPse2eandSAF0DGMQm
  app: app_name
  on:
    branch: master
    python: '3.6'
    repo: github_username/repo_name
```

Update this, commit, and then push to Github. Since we're still working off the Master branch, this will trigger a deploy.

Testing Branch

Let's test out the Feature Branch workflow.

Start by creating a new branch:

```
$ git checkout -b unit-tests master
```

This command creates a new branch called `unit-tests` based on the code in the current Master branch.

NOTE: You can tell which branch you're currently working on by running: `git branch`.

Since this is the testing branch, let's add another unit test to `test_main.py`:

```
def test_index(self):  
    """ Ensure flask was set up correctly. """  
    response = self.app.get('/', content_type='html/text')  
    self.assertEqual(response.status_code, 200)
```

Run the tests. Commit the changes locally. Then push to Github.

```
$ git add -A  
$ git commit -am "added tests"  
$ git push origin unit-tests
```

(Or use the fabfile: `fab prepare`)

Notice how we pushed the code to the Feature branch rather than to Master. Now after Travis CI runs, and the tests pass, the process ends. The code is *not* deployed since we are not working off the Master branch.

Add more tests. Commit. Push. Repeat as much as you'd like. Try committing your code in logical batches - e.g., after you are finished testing a specific function, commit your code, and push to the Feature branch on Github. This is exactly what we talked about in the beginning of the chapter - integrating your code many times a day.

When you're done testing, you're now ready to merge the Feature branch into Master.

Create a Pull Request

Follow the steps [here](#) to create a pull request. Notice how Travis CI will add a new build to the queue. Once the tests pass, click the "Merge Pull Request" button, and then "Confirm Merge". Now the tests run again, but this time, since they're running against the Master branch, a deployment will happen as well.

Be sure to review the steps that you went through for the Feature Branch Workflow before moving on.

How about a quick sanity check! Comment out the following code:

```
@users_blueprint.route('/', methods=['GET', 'POST'])
def login():
    error = None
    form = LoginForm(request.form)
    if request.method == 'POST':
        if form.validate_on_submit():
            user = User.query.filter_by(name=request.form['name']).first()
            if user is not None and bcrypt.check_password_hash(
                user.password, request.form['password']):
                session['logged_in'] = True
                session['user_id'] = user.id
                session['role'] = user.role
                session['name'] = user.name
                flash('Welcome!')
                return redirect(url_for('tasks.tasks'))
            else:
                error = 'Invalid username or password.'
    return render_template('login.html', form=form, error=error)
```

Commit locally. Push to Github. Create then merge the pull request. What happened? Although Travis CI ran the tests against the Master branch, since the build failed it did *not* deploy the code, which is exactly what we want to happen. Fix the code before moving on.

What's next?

Fabric

Remember our [Fabric](#) script? We can still use it here to automate the CI/CD processes.

Update the `push()` function like so:

```
def push():
    local("git branch")
    branch = raw_input("Which branch do you want to push to? ")
    local("git push origin {}".format(branch))
```

Now we can test the functionally:

```
$ fab prepare
fab prepare
[localhost] local: nosetests -v
Enter a git commit message: fabric test
[localhost] local: git add -A && git commit -am 'fabric test'
[unit-tests 5142eff] fabric test
 3 files changed, 8 insertions(+), 29 deletions(-)
[localhost] local: git branch
  master
* unit-tests
Which branch do you want to push to? unit-tests
[localhost] local: git push origin unit-tests
Counting objects: 9, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 611 bytes, done.
Total 5 (delta 4), reused 0 (delta 0)
To git@github.com:realpython/flasktaskr_project.git
  c1d508c..5142eff  unit-tests -> unit-tests

Done.
```

This committed our code locally, then pushed the changes up to Github on the Feature branch. As you know, this triggered Travis-CI, which ran our tests. Next, manually go to the Github repository and create the pull request, wait for the tests to pass on Travis CI, and then merge the pull request into the Master branch. This, in turn, will trigger another round of automated tests. Once these pass, the code will be deployed to Heroku.

Recap

Let's look at workflow (again!) in terms of our current solution:

Step	Details	Action
1	Create a new branch locally	<code>git checkout -b <feature></code> <code>git checkout master</code>
2	Develop locally	Update your code
3	Commit locally, then push to Github	<code>fab prepare</code>
4	Run the automated tests	Triggered automatically by Travis CI
5	Repeat until the feature is complete	Repeat steps 2 through 4
6	Open a pull request against the Master branch	Follow the process outlined here
7	Run the automated tests	Triggered automatically by Travis CI
8	If the tests pass, deploy to Heroku	Triggered automatically by Travis CI

Conclusion

And with that, we are done with Flask for now.

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	Yes
Testing	Yes
Styling	Yes
Test Coverage	Yes
Nose Testing Framework	Yes
Permissions	Yes
Blueprints	Yes
New Features	Yes
Password Hashing	Yes
Custom Error Pages	Yes
Error Logging	Yes
Deployment Options	Yes
Automated Deployments	Yes
Building a REST API	Yes
Boilerplate and Workflow	Yes
Continuous Integration and Delivery	Yes

We will be revising this app again in the future where we'll-

- Upgrade to Postgres,
- Add jQuery and AJAX,
- Update the app configuration,
- Utilize Flask-Script and Flask-Migrate,
- Expand the RESTful API, and
- Add integration tests.

Patience. Keep practicing.

With that, let's move on to Behavior Driven Development!

Flask: Behavior-Driven Development with Behave

The goal of this chapter is to introduce you to a powerful technique for developing and testing your web applications called [behavior-driven development](#) (BDD). The application we're going to build is simple yet complex enough for you to learn how powerful BDD can be.

As you probably know, [Flaskr](#) - a mini-blog-like-app - is the app you build for the official tutorial for Flask. Well, we'll be taking the tutorial a step further by developing it using the BDD paradigm.

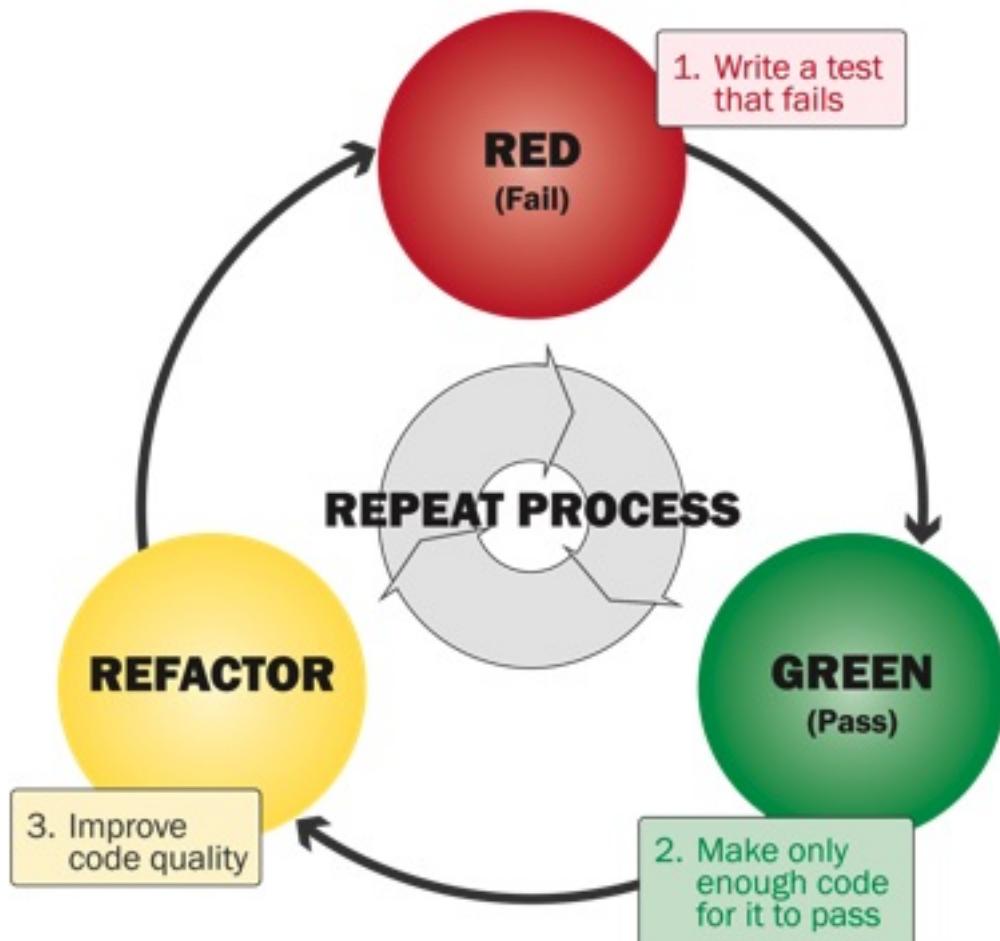
Enjoy.

Behavior-Driven Development

BDD is a process used to drive out the functionality of your application by focusing on user behavior - or how the end user wants the application to behave.

Based on the principles of test-driven development (TDD), it actually extends TDD in that test cases are written in natural language that any stakeholder can read and (hopefully) understand. Since it is an extension of TDD, BDD emphasizes writing automated tests before writing the actual feature or functions as well as the "Red-Green-Refactor" cycle, as shown in the image below:

1. Write a test
2. Run all the tests (the new test should fail)
3. Write just enough code to get the new test to pass
4. Refactor the code (if necessary)
5. Repeat steps 1 through 4



TDD can often lead to more modularized, flexible, and extensible code; the frequent nature of testing helps to catch defects early in the development cycle, preventing them from becoming large, expensive problems.

NOTE: Remember that theory is much, much different than practice. In most production environments, there are simply not enough resources to test everything and often testing is pushed off until the last minute - e.g., when things break.

That said, the BIG difference between TDD and BDD is that before writing any tests, you first write the actual feature that is being tested in natural language.

For example:

```
Scenario: successful login
  Given flaskr is set up
    When we log in with "admin" and "admin"
      Then we should see the alert "You were logged in"
```

Once that feature is defined, you then write tests based on *actions* and the expected *behaviors* of the feature. In this case, we'd test that the application (Flaskr) is set up and that after a successful log in, the text "You were logged in" is displayed.

Thus the new BDD cycle looks like this:

1. Add the feature file
2. Write a step (or test)
3. Run all the tests (the new test should fail)
4. Write just enough code to get the new test to pass
5. Refactor the code (if necessary)
6. Repeat steps 2 through 5, until all scenarios are tested for
7. Start again at step 1

Done right, using BDD, you can answer some of the hardest questions that plague developers of all skill levels:

1. Where do I begin to develop?
2. What exactly should I write tests for?
3. How should my tests be structured??

Homework

- Before we start, take a minute to read more about behavior-driven development [here](#) along with this Intro to BDD [tutorial](#).
- Want to learn more about TDD? Check out this [tutorial](#), which details building the Flaskr app using the TDD paradigm. This is optional, but helpful, especially if you've never practiced TDD before.

Project Setup

Test your skills by setting up basic a "Hello World" application. Try to do as much of this as you can without looking back at the *Flask Quick Start* chapter. Don't worry if you can't figure out everything on your own. As long as the process is a little easier and you understand it a little better, then you are learning. One step at a time.

Start with this:

1. Within your "realpython" directory create a "flask-bdd" directory.
2. Create and activate your virtual environment.

Set up the following files and directories:

```
└── app.py
└── static
    ├── css
    ├── img
    └── js
└── templates
```

Now build your app. Although there are a number of ways to code this out, once complete, make sure to copy my code over from the "assets" directory in the Real Python Exercises [repo](#).

Finally, there are three more steps:

1. Rename the `app.py` file to `flaskr.py`
2. Run a sanity check; fire up the server, then make sure the app is working correctly.
3. Install Behave - `pip install behave==1.2.5`

Introduction to Behave

[Behave](#) is a fun (yes, fun) tool used for writing automated **acceptance tests** to support development in the BDD style. Not only is it one of the most popular Python-based behavior-driven tools, it also is easy to learn and use. It utilizes a domain-specific, human readable language named **Gherkin** to allow the execution of feature documentation written in natural language text. These features are then turned into test code written in Python.

Behave bridges non-developer stakeholders (such as project managers and external customers) and development teams, helping to eliminate misunderstandings and technical waste.

Let's dive right in.

Homework

- Complete the basic tutorial from the Behave [documentation](#), making sure that you understand the relationship between Features, Scenarios, and Steps.

Feature Files

Feature files, which are written in natural language, define where in your application the tests are supposed to cover. This not only helps with thinking through your app, but it also makes documenting much easier.

Flaskr

Start by looking at the [features](#) for the Flaskr App:

1. Let the user sign in and out with credentials specified in the configuration. Only one user is supported.
2. When the user is logged in, they can add new entries to the page consisting of a text-only title and some HTML for the text. This HTML is not sanitized because we trust the user here.
3. The page shows all entries so far in reverse order (newest on top), and the user can add new ones from there if logged in.

We can use each of the above features to define our feature files for Behave.

Step back for a minute and pretend you're working with a client on a project. If you're given a list of high-level features, you can immediately build your feature files, tying in scenarios to help drive out the main functions of the application. Since this is all completed in natural language, you can return to your client to work through each feature file and make changes as needed. This accomplishes several things, most notability - accountability.

You're working closer with your client (or project manager, etc.) to define through user behavior the functions of the application. Both parties are on the same page, which helps to decrease the gap between what's expected and what's ultimately delivered.

First Feature

Let the user sign in and out with credentials specified in the configuration. Only one user is supported.

First, create a new directory called "features", which will eventually house all of our features files as well the subsequent tests, called steps. Within that folder, create a file called `auth.feature` (no extra file extension) and add the following text, which is written in a language/style called [Gherkin](#):

```
Feature: flaskr is secure in that users must log in and log out to access certain
features

  Scenario: successful login
    Given flaskr is set up
    When we log in with "admin" and "admin"
    Then we should see the alert "You were logged in"

  Scenario: incorrect username
    Given flaskr is set up
    When we log in with "notright" and "admin"
    Then we should see the alert "Invalid username"

  Scenario: incorrect password
    Given flaskr is set up
    When we log in with "admin" and "notright"
    Then we should see the alert "Invalid password"

  Scenario: successful logout
    Given flaskr is set up
    and we log in with "admin" and "admin"
    When we log out
    Then we should see the alert "You were logged out"
```

The actual feature is the main story, then each scenario is like a separate chapter. We now need to write our tests, which are called steps, based on the scenarios.

Create a "steps" directory within the "features" directory, then add a file called `auth_steps.py`. Let's add our first test:

```
from behave import *

@given(u'flaskr is set up')
def flask_is_setup(context):
    assert context.client
```

First, notice that `flaskr is set up` matches the text within the Feature file. Next, do you know what we're testing here? Do you know what a request context is? If not, please read this [article](#).

Run Behave:

```
$ behave
```

Search through the output until you find the overall stats on what features and scenarios fail:

```
Failing scenarios:
  features/auth.feature:3  successful login
  features/auth.feature:8  incorrect username
  features/auth.feature:13  incorrect password
  features/auth.feature:18  successful logout

  0 features passed, 1 failed, 0 skipped
  0 scenarios passed, 4 failed, 0 skipped
  0 steps passed, 4 failed, 0 skipped, 9 undefined
  Took 0m0.001s
```

Since this test failed, we now want to *write just enough code to get the new test to pass*.

Also, did you notice that only four steps failed - `0 steps passed, 4 failed, 0 skipped, 9 undefined`. How can that be if we only defined one step? Well, that step actually appears four times in our feature file. Open the file and find them. *Hint*: Look for all instances of `flaskr is set up`.

Environment Control

If you went through the Behave tutorial, which you should have already done, then you know that you had to set up an `environment.py` file to specify certain `tasks` to run before and after each test. We need to do the same thing.

Create an `environment.py` file within your "features" folder, and then add the following code:

```
import os
import sys

## add module to syspath
# get the current working directory
cwd = os.path.abspath(os.path.dirname(__file__))
# isolate the last folder (the folder we are currently in)
project = os.path.basename(cwd)
# remove the last folder from the cwd
new_path = cwd.strip(project)
# create a new path pointing to where our Flask object is defined
full_path = os.path.join(new_path, 'flaskr')

try:
    from flaskr import app
except ImportError:
    sys.path.append(full_path)
    from flaskr import app

def before_feature(context, feature):
    context.client = app.test_client()
```

What's going on?

First, take a look at the function. `Flask` provides a handy method, `test_client()`, to use during testing to essentially mock the actual application.

In order for this function to run correctly - e.g., in order to mock our app - we need to provide an instance of our actual Flask app. How do we do that? Well, first ask yourself: "Where did we establish an instance of Flask to create our app?" Of course! It's within `flaskr.py`:

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

We need access to that variable.

We can't just import that app because right now we do not have access to that script in the current directory. Thus, we need to add that script to our PATH, using

```
sys.path.append :
```

```
sys.path.append(full_path)
```

Yes, this is confusing. Basically, when the Python interpreter sees that a module has been imported, it tries to find it by searching in various locations - and the PATH is one of those locations. Read more about this from the official Python [documentation](#).

Run Behave again.

```
Failing scenarios:
  features/auth.feature:3  successful login
  features/auth.feature:8  incorrect username
  features/auth.feature:13  incorrect password
  features/auth.feature:18  successful logout

0 features passed, 1 failed, 0 skipped
0 scenarios passed, 4 failed, 0 skipped
4 steps passed, 0 failed, 0 skipped, 9 undefined
Took 0m0.000s
```

This time, all of our features and scenarios failed, but four steps passed. Based on that, we know that the first step passed. Why? Remember that four steps failed when we first ran the tests. You can also check the entire stack trace to see the steps that passed (highlighted in green).

Next steps

Update the *auth_steps.py* file:

```

@given(u'flaskr is set up')
def flask_is_set_up(context):
    assert context.client

@given(u'we log in with "{username}" and "{password}"')
@when(u'we log in with "{username}" and "{password}"')
def login(context, username, password):
    context.page = context.client.post(
        '/login',
        data=dict(username=username, password=password),
        follow_redirects=True
    )
    assert context.page

@when(u'we log out')
def logout(context):
    context.page = context.client.get('/logout', follow_redirects=True)
    assert context.page

@then(u'we should see the alert "{message}"')
def message(context, message):
    assert str.encode(message) in context.page.data

```

Compare this to your feature file to see the relationship between scenarios and steps. Run Behave. Take a look at the entire stack trace in your terminal. Notice how the variables in the step file - i.e., `{username}` , are replaced with the provided username from the feature file.

NOTE: We are breaking a rule by writing more than one test in a single iteration of the BDD cycle. We're doing this for time's sake. Each of these tests are related and some even overlap, as well. That said, when you go through this process on your own, I highly recommend sticking with one test at a time. In fact, if you are feeling ambitious, deviate from this guide and try writing a single step at a time, going through the BDD process, until all steps, scenarios, and features pass.

Then move on to the Second Feature...

What's next? Building the log in and log out functionality into our code base. Remember: Write *just enough* code to get this it pass. Want to try on your own before looking at my answer?

Login and Logout

Update `flaskr.py`

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    """User login/authentication/session management."""
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
            error = 'Invalid username'
        elif request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('index'))
    return render_template('login.html', error=error)

@app.route('/logout')
def logout():
    """User logout/authentication/session management."""
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('index'))
```

Based on the route decorator, the `login()` function can either accept a GET or POST request. If the method is a POST, then the provided username and password are checked. If both are correct, the user is logged in then redirected to the main, `/`, route; and a key is added the session as well. But if the credentials are incorrect, the login template is re-rendered along with an error message.

What happens in the `logout()` function? Need [help](#)?

Update the config

```
# imports
from flask import Flask, render_template, request, session

# configuration
DATABASE = ''
USERNAME = 'admin'
PASSWORD = 'admin'
SECRET_KEY = 'change me'
```

Update the templates

Add the `login.html` template:

```
<div class="container">
  <h1>Flask - BDD</h1>
  {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
  {% endfor %}
  <h3>Login</h3>
  {% if error %}<p class="error"><strong>Error:</strong> {{ error }}{% endif %}
  <form action="{{ url_for('login') }}" method="post">
    <dl>
      <dt>Username:</dt>
      <dd><input type="text" name="username"></dd>
      <dt>Password:</dt>
      <dd><input type="password" name="password"></dd>
      <br>
      <br>
      <dd><input type="submit" class="btn btn-default" value="Login">
        <span>Use "admin" for the username and password</span>
      </dd>
    </dl>
  </form>
</div>
```

While we're at it, let's update the styles; update the head of *base.html*:

```
<head>
  <title>Flask - Behavior Driven Development</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="http://netdna.bootstrapcdn.com/
    bootstrap/3.1.1/css/bootstrap.min.css" rel="stylesheet" media="screen">
  <script src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
  <script src="http://netdna.bootstrapcdn.com/
    bootstrap/3.1.1/js/bootstrap.min.js"></script>
</head>
```

Yes, we broke another rule. We are writing a bit more code than necessary. It's okay, though. This happens in the development process. TDD and BDD are just guides to follow, providing you with a path to (hopefully) better the development process as a whole. Every now and then you'll have to bend or break the rules. Now is a good time.

Finally, update *index.html*:

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

{% block content %}

<div class="container">
  <h1>Flask - BDD</h1>
  {% if not session.logged_in %}
    <a href="{{ url_for('login') }}>log in</a>
  {% else %}
    <a href="{{ url_for('logout') }}>log out</a>
  {% endif %}
  {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
  {% endfor %}
  {% if session.logged_in %}
    <h1>Hi!</h1>
  {% endif %}
</div>

{% endblock %}
```

Sanity check

Now, before we run Behave, let's do some manual testing. Run the server. Test logging in and logging out. Did you remember to add the right imports? Are the right messages flashed? Compare the actual behavior with the expected behavior in the feature and step files. Are you confident that they align? If so, run Behave:

```
1 feature passed, 0 failed, 0 skipped
4 scenarios passed, 0 failed, 0 skipped
13 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.208s
```

Nice! One feature down. Three more to go.

Second Feature

When the user is logged in, they can add new entries to the page consisting of a text-only title and some HTML for the text. This HTML is not sanitized because we trust the user here.

What are the steps for BDD again?

1. Add the feature file
2. Write a step (or test)
3. Run all the tests (the new test should fail)
4. Write just enough code to get the new test to pass
5. Refactor the code (if necessary)
6. Repeat steps 2 through 5, until all scenarios are tested for
7. Start again at step 1

Add (err, update) the feature file

Add the following scenarios to `auth.feature`:

```
Scenario: successful post
  Given flaskr is set up
  and we log in with "admin" and "admin"
    When we add a new entry with "test" and "test" as the title and text
    Then we should see the alert "New entry was successfully posted"

Scenario: unsuccessful post
  Given flaskr is set up
  Given we are not logged in
    When we add a new entry with "test" and "test" as the title and text
    Then we should see the alert "Unauthorized"
```

This should be fairly straightforward. Although, given these scenarios, it can be hard to write the steps. Let's focus on one at a time.

NOTE: We could also create an entire new feature file for this. However, since there is much overlap in the code and the auth features are really testing *all* of the user behavior for when a user is logged in vs. logged out, it's okay to add this to the auth feature file. If there were a number of actions or behaviors that a logged in user could perform, then, yes, it would be best to separate this out into a new feature file. Just be aware of when you are writing the same code over and over again; this should trigger a [code smell](#).

First Step

Add the following step to `auth.steps`:

```
@when(u'we add a new entry with "{title}" and "{text}" as the title and text')
def add(context, title, text):
    context.page = context.client.post(
        '/add',
        data=dict(title=title, text=text),
        follow_redirects=True
    )
    assert context.page

@given(u'we are not logged in')
def logout(context):
    context.page = context.client.get('/logout', follow_redirects=True)
```

Run Behave

```
Failing scenarios:
  features/auth.feature:24  successful post
  features/auth.feature:30  unsuccessful post

0 features passed, 1 failed, 0 skipped
4 scenarios passed, 2 failed, 0 skipped
17 steps passed, 1 failed, 1 skipped, 2 undefined
Took 0m0.050s
```

Let's get this scenario, `successful post`, to pass.

Update `flaskr.py`

```
@app.route('/add', methods=['POST'])
def add_entry():
    flash('New entry was successfully posted')
    return redirect(url_for('index'))
```

Update *index.html*

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

{% block content %}

<div class="container">
    <h1>Flask - BDD</h1>
    {% if not session.logged_in %}
        <a href="{{ url_for('login') }}>log in</a>
    {% else %}
        <a href="{{ url_for('logout') }}>log out</a>
    {% endif %}
    {% for message in get_flashed_messages() %}
        <div class="flash">{{ message }}</div>
    {% endfor %}
    {% if session.logged_in %}
        <br><br>
        <form action="{{ url_for('add_entry') }}" method="post" class="add-entry">
            <dl>
                <dt>Title:
                <dd><input type="text" size="30" name="title">
                <dt>Text:
                <dd><textarea name="text" rows="5" cols="40"></textarea>
                <br>
                <br>
                <dd><input type="submit" class="btn btn-default" value="Share">
            </dl>
        </form>
    {% endif %}
</div>

{% endblock %}
```

Run Behave

```
Failing scenarios:
  features/auth.feature:30  unsuccessful post

0 features passed, 1 failed, 0 skipped
5 scenarios passed, 1 failed, 0 skipped
18 steps passed, 0 failed, 1 skipped, 2 undefined
Took 0m0.325s
```

Nice. The `successful post` scenario passed. Moving on to the next scenario, `unsuccessful post` ...

Update `flaskr.py`

We just need to add a conditional to account for users not logged in:

```
@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('logged_in'):
        abort(401)
    flash('New entry was successfully posted')
    return redirect(url_for('index'))
```

Be sure to update the imports as well:

```
# imports
from flask import (Flask, render_template, request,
    session, flash, redirect, url_for, abort)
```

Run Behave

Looks good!

```
1 feature passed, 0 failed, 0 skipped
6 scenarios passed, 0 failed, 0 skipped
21 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.218s
```

On to the final feature...

Third Feature

The page shows all entries so far in reverse order (newest on top) and the user can add new ones from there if logged in.

Remember:

1. Add the feature file
2. Write a step (or test)
3. Run all the tests (the new test should fail)
4. Write just enough code to get the new test to pass
5. Refactor the code (if necessary)
6. Repeat steps 2 through 5, until all scenarios are tested for
7. Start again at step 1

Update the feature file

Again, update the current feature file by adding another "Then" to the end of the "successful post" scenario:

```
Scenario: successful post
  Given flaskr is set up
  and we log in with "admin" and "admin"
    When we add a new entry with "test" and "test" as the title and text
    Then we should see the alert "New entry was successfully posted"
    Then we should see the post with "test" and "test" as the title and text
```

First Step

```
@then(u'we should see the post with "{title}" and "{text}" as the title and text')
def entry(context, title, text):
    assert title and text in context.page.data
```

This should be clear by now. We're simply checking that the posted title and text are found on the page after the logged in user submits a new post.

Run Behave

```
Failing scenarios:
  features/auth.feature:24  successful post

0 features passed, 1 failed, 0 skipped
5 scenarios passed, 1 failed, 0 skipped
21 steps passed, 1 failed, 0 skipped, 0 undefined
Took 0m0.058s
```

This failed as expected. Now let's get the step to pass.

Create a new file called **schema.sql**:

```
drop table if exists entries;
create table entries (
    id integer primary key autoincrement,
    title text not null,
    text text not null
);
```

Here we are setting up a single table with three fields - "id", "title", and "text".

Update **flaskr.py**

Add two new imports, `sqlite3` and `g`:

```
import sqlite3
from flask import (Flask, render_template, request,
    session, flash, redirect, url_for, abort, g)
```

Update the config section:

```
# configuration
DATABASE = 'flaskr.db'
USERNAME = 'admin'
PASSWORD = 'admin'
SECRET_KEY = 'change me'
```

Add the following functions for managing the database connections:

```
# connect to database
def connect_db():
    rv = sqlite3.connect(app.config['DATABASE_PATH'])
    rv.row_factory = sqlite3.Row
    return rv

# create the database
def init_db():
    with app.app_context():
        db = get_db()
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()

# open database connection
def get_db():
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db

# close database connection
@app.teardown_appcontext
def close_db(error):
    if hasattr(g, 'sqlite_db'):
        g.sqlite_db.close()
```

Finally, update the `index()` and `add_entry()` functions:

```
@app.route('/')
def index():
    db = get_db()
    cur = db.execute('select title, text from entries order by id desc')
    entries = cur.fetchall()
    return render_template('index.html', entries=entries)

@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('logged_in'):
        abort(405)
    db = get_db()
    db.execute('insert into entries (title, text) values (?, ?)',
               [request.form['title'], request.form['text']])
    db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('index'))
```

The `index()` function queries the database, which we still need to create, grabbing all the entries, displaying them in reverse order so that the newest post is on top. *Do we have a step for testing to ensure that new posts are displayed first?* The `add_entry()` function now adds the data from the form to the database.

Create the database

```
$ python
>>> from flaskr import init_db
>>> init_db()
>>>
```

Update `index.html`

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

{% block content %}

<div class="container">
<h1>Flask - BDD</h1>
{% if not session.logged_in %}
<a href="{{ url_for('login') }}>log in</a>
{% else %}
<a href="{{ url_for('logout') }}>log out</a>
{% endif %}
{% for message in get_flashed_messages() %}
<div class="flash">{{ message }}</div>
{% endfor %}
{% if session.logged_in %}
<br><br>
<form action="{{ url_for('add_entry') }}" method="post" class="add-entry">
<dl>
<dt>Title:
<dd><input type="text" size="30" name="title">
<dt>Text:
<dd><textarea name="text" rows="5" cols="40"></textarea>
<br>
<br>
<dd><input type="submit" class="btn btn-default" value="Share">
</dl>
</form>
{% endif %}
<ul>
{% for entry in entries %}
<li><h2>{{ entry.title }}</h2>{{ entry.text|safe }}</li>
{% else %}
<li><em>No entries yet. Add some!</em></li>
{% endfor %}
</ul>
</div>

{% endblock %}
```

This adds a loop to loop through each entry, posting the title and text of each.

Environment Control

Update:

```

import os
import sys
import tempfile

# add module to syspath
pwd = os.path.abspath(os.path.dirname(__file__))
project = os.path.basename(pwd)
new_path = pwd.strip(project)
full_path = os.path.join(new_path, 'flaskr')

try:
    from flaskr import app, init_db
except ImportError:
    sys.path.append(full_path)
    from flaskr import app, init_db

def before_feature(context, feature):
    app.config['TESTING'] = True
    context.db, app.config['DATABASE'] = tempfile.mkstemp()
    context.client = app.test_client()
    init_db()

def after_feature(context, feature):
    os.close(context.db)
    os.unlink(app.config['DATABASE'])

```

Now we're setting up our database for test conditions by creating a completely new database and assigning it to a temporary file using the `tempfile.mkstemp()` function. After each feature has run, we close the database context and then switch the primary database used by our app back to the one defined in `flaskr.py`.

We should now be ready to test again.

Run Behave

```

1 feature passed, 0 failed, 0 skipped
6 scenarios passed, 0 failed, 0 skipped
22 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.307s

```

All clear!

Update Steps

There's still plenty of functionality that we missed. In some cases we need to update the scenarios, while in others we can just update the steps. For example, we should probably update the following step so that we're ensuring the database is set up as well as the app itself:

```
@given(u'flaskr is setup')
def flask_is_set_up(context):
    assert context.client and context.db
```

Go ahead and make the changes, then re-run Behave.

Other changes:

1. Test to ensure that posts are displayed in reverse order.
2. When a user is not logged in, posts are still displayed.
3. Regardless of being logged in or not, if there are no posts in the database, then the text "No entries yet. Add some!" should be found on the page.

See if you can implement these on your own. Have fun!

Conclusion

Besides Behave, there's only a few other BDD frameworks for Python, all of which are immature (even Behave, for that matter). If you're interested in trying out other frameworks, the people behind Behave put an excellent [article](#) together that compares each BDD framework. It's worth a read.

Despite the immature offerings, BDD frameworks provide a means of delivering useful acceptance, unit, and integration tests. Integration tests are designed to ensure that all units (or scenarios) work well together. This, coupled with the focus on end user behavior, helps guarantee that the final product bridges the gap between business requirements and actual development to meet user expectations.

Homework

- Take the quick [Python Developer Test](#).
- Want even more Flask? Watch all videos from the [Discover Flask](#) series. Note: New videos are still being added.

Interlude: Web Frameworks, Compared

As previously mentioned, web frameworks alleviate the overhead incurred from common, repetitive tasks associated with web development. By using a web framework, web developers delegate responsibility of low-level tasks to the framework itself, allowing the developer to focus on the application logic.

These low-level tasks include handling of:

- Client requests and subsequent server responses
- URL routing
- Separation of concerns (application logic vs. HTML output)
- Database communication

NOTE: Take note of the concept "Don't Repeat Yourself" (or DRY). Always avoid reinventing the wheel. This is exactly what web frameworks excel at. The majority of the low-level tasks (listed above) are common to every web application. Since frameworks automate much of these tasks, you can get up and running quickly, so you can focus your development time on what really matters: focusing on the product and making your application stand out from the crowd.

Most frameworks also include a development web server, which is a great tool used not only for rapid development but automating testing as well.

The majority of web frameworks can be labeled as either a full (high-level), or micro (low-level) framework, depending on the amount and level of automation it performs and the number of pre-installed components (batteries) it comes with. Full frameworks come with many pre-installed batteries and a number of low-level task automation, while micro frameworks come with few batteries and less automation.

Since all web frameworks offer some automation to help speed up web development, in the end, it's up to the developer to decide how much control s/he wants. Beginning developers should first focus on demystifying much of the *magic*, (commonly referred to as automation), to help understand the differences between the various web frameworks and avoid later confusion.

Keep in mind that while there are plenty of upsides to using web frameworks, most notably rapid development, there is also a huge downside: lock-in. Put simply, by choosing a specific framework, you lock your application into that framework's

philosophy and become dependent on the framework's developers to maintain and update the code base.

In general, problems are more likely to arise with high-level frameworks due to the automation and features they provide; you have to do things *their* way. However, with low-level frameworks, you have to write more code up front to make up for the missing features, which slows the development process in the beginning. There's no right answer, but you do not want to have to change a mature application's framework; this is a daunting task to say the least. Choose wisely.

Overview

Popular Frameworks

1. **web2py, Django, and Turbogears** are all full frameworks, which offer a number of pre-installed utilities and automate many tasks beneath the hood. They all have excellent documentation and community support. The high-level of automation, though, can make the learning curve for these quite steep.
2. **web.py, CherryPy, and bottle.py**, are micro-frameworks with few 'batteries' included, and automate only a few underlying tasks. Each has excellent community support and are easy to install and work with. web.py's documentation is a bit unorganized, but still relatively well-documented like the other frameworks.
3. Both **Pyramid** and **Flask**, which are still considered micro-frameworks, have quite a few pre-installed 'batteries' and a number of additional pre-configured 'batteries' available as well, which are very easy to install. Again, documentation and community support are excellent, and both are very easy to use.

Components

Before starting development with a new framework, learn what pre-installed and available 'batteries'/libraries it offers. At the core, most of the components work in a similar manner; however, there are subtle differences. Take Flask and web2py, for example; Flask uses an ORM for database communication and a type of template engine called Jinja2. web2py, on the other hand, uses a DAL for communicating with a database and has its own brand of templates.

Don't just jump right in, in other words, thinking that once you learn to develop in one, you can use the same techniques to develop in another. Take the time to learn the differences between frameworks to avoid later confusion.

What does this all mean?

As stated in the Introduction of this course, the framework(s) you decide to use should depend more on which you are comfortable with and your end-product goals. If you try various frameworks and learn to recognize their similarities while also respecting their differences, you will find a framework that suits your tastes as well as your application.

Don't let other developers dictate what you do or try. Find out for yourself!

web2py: Quick Start

web2py is a high-level, open source web framework designed for rapid development. With web2py, you can accomplish everything from installation, to project setup, to actual development, quickly and easily. In no time flat, you'll be up and running and ready to build beautiful, dynamic websites.



In this section, we'll be exploring the fundamentals of web2py - from installation to the basic development process. You'll see how web2py automates much of the low-level, routine tasks of development, resulting in a smoother process, and one which allows you to focus on high-level issues. web2py also comes pre-installed with many of the components we had to manually install for Flask.

web2py, developed in 2007 by [Massimo Di Pierro](#) (an associate professor of Computer Science at DePaul University), takes a different approach to web development than the other Python frameworks. Because Di Pierro [aimed](#) to lower the barrier for entry into web development, so more automation (often called *magic*) happens under the hood, which can make development simpler, but it can also give you less control over how your app is developed. All frameworks share a number of basic common traits. If you learn these as well as the web development fundamentals from Section One, you will better understand what's happening behind the scenes, and thus know how to make changes to the automated processes for better customization.

Homework

- Watch [this](#) excellent speech by Di Pierro from PyCon US 2012. Don't worry if the concepts don't make sense right now. They will soon enough. Pause the video at times and look up any concepts that you don't understand. Take notes.

Installation

Quick Install

NOTE: This course utilizes web2py version [2.14.6](#).

If you want to get [started](#) quickly, you can [download](#) the binary archive, unzip, and run either web2py.exe (Windows) or web2py.app (Unix). You must set an administrator password to access the administrative interface. The Python Interpreter is included in the archive, as well as many third party libraries and packages. You will then have a development environment set up, and be ready to start building your application - all in less than a minute, and without even having to touch the terminal.

We'll be utilizing a different workflow by developing from the command line, so follow the full installation guide below.

Full Install

Create a directory called "web2py". To clone the git repo from the web2py [repository](#) run these commands from your *real-python* directory:

```
$ git clone https://github.com/web2py/web2py.git --recursive
```

You should see something like this:

```
Cloning into 'web2py'...
remote: Counting objects: 36290, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 36290 (delta 0), reused 0 (delta 0), pack-reused 36283
Receiving objects: 100% (36290/36290), 36.94 MiB | 10.00 MiB/s, done.
Resolving deltas: 100% (22648/22648), done.
Checking connectivity... done.
Submodule 'gluon/packages/dal' (https://github.com/web2py/pydal.git) registered for path 'gluon/packages/dal'
Cloning into 'gluon/packages/dal'...
remote: Counting objects: 32193, done.
remote: Total 32193 (delta 0), reused 0 (delta 0), pack-reused 32193
Receiving objects: 100% (32193/32193), 30.82 MiB | 8.58 MiB/s, done.
Resolving deltas: 100% (19633/19633), done.
Checking connectivity... done.
Submodule path 'gluon/packages/dal': checked out '12bc6d97402acce462c1193f57bbba4afde7c3c3'
```

NOTE: The `--recursive` flag is important in this command. When this command executes, it will automatically install the dependencies we need to run `web2py`.

Then `cd` into that directory:

```
$ cd web2py
```

NOTE: Both apps within this chapter will be developed within this same instance of `web2py`.

Ok great! We have `web2py` on our machine and it is accessible through the command line.

As of writing (October 18, 2016), `web2py` only works with Python 2.7.x. Because of this, we will need a new command line tool to create our environment, as the `'python3 -m venv'` command was just introduced in Python 3.6.

We will use [pyenv](#). For a great tutorial on virtual environments in python checkout [this blog post](#). The post talks about 'pyenv' toward the end.

Lets install 'pyenv':

```
$ brew install pyenv
```

NOTE: If you don't have Homebrew installed yet go [here](#). Depending on where (what directory) you have Python installed on your machine you may need to also install both Python 2.7 and Python 3.6 via Homebrew so that everything is wired up properly.

Install Python 2.7 and 3.6.0 via 'pyenv':

```
$ pyenv install 2.7
$ pyenv install 3.6
```

Now, checkout what versions of Python we have installed:

```
$ pyenv versions
```

The output should be something like this:

```
web2py$ pyenv versions
* system (set by /Users/michaelherman/.pyenv/version)
  2.7
  3.6
```

NOTE: All of this installation has been *global* on your computer. That means you will have access to 'pyenv' in any of your directories. What we are going to do next will be local to the your current working directory. In our case *web2py/*.

So now, set the Python version that will be specific to this directory:

```
$ pyenv local 2.7
```

Then, back on your command line launch *web2py*:

```
$ pyenv exec python web2py.py
```

This command runs the *web2py.py* file using the 'local' version of python that we just set, 2.7.

After *web2py* loads, set an admin password, and you're good to go. FYI, there is no need to activate and deactivate your virtual environments when using pyenv. But you do need to run your shell commands with `pyenv exec` in order to maintain your project environment.

NOTE: web2py by default separates projects (a collection of related applications); however, it's still important to use a pyenv to keep your third-party libraries isolated from one another. That said, we'll be using the same web2py instance and directory, "start", for the apps in this chapter.

Regardless of how you installed web2py (Quick vs Full), a number of libraries are pre-imported. These provide a functional base to work with so you can start programming dynamic websites as soon as web2py is installed. We'll be addressing such libraries as we go along.

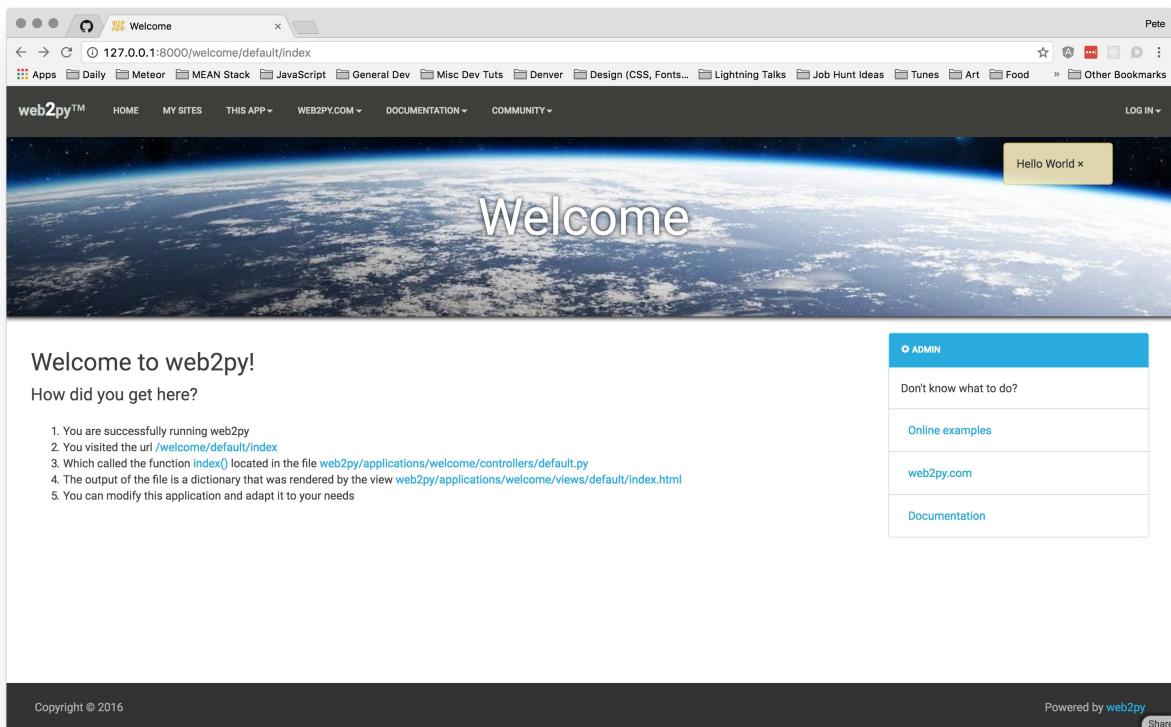
Hello World

Fire up the server:

```
$ pyenv exec python web2py.py
```

Input your admin password taking you to the *web2py* application home page.

Click the link labeled `admin` and then input your admin password.



You're now on the **Sites** page. This is the main administration page where you create and modify your applications.

To create a new application, type the name of the app, "hello_world", in the text field below "New simple application". Click create to be taken to the **Edit** page. All new applications are just copies of the "welcome" app:

NOTE You'll soon find out that web2py has a default for pretty much everything (but it can be modified). In this case, if you don't make any changes to the views, for example, your app will have the basic styles and layout taken from the default, "welcome" app.

Think about some of the pros and cons to having a default for everything. You obviously can get an application set up quickly. Perhaps if you aren't adept at a particular part of development, you could just rely on the defaults. However, if you do go that route you probably won't learn anything new - and you could have a difficult transition to another framework that doesn't rely as much (or at all) on defaults. Thus, I urge you to practice. Break things. Learn the default behaviors. Make them better. Make them your own.

The **Edit** page is logically organized around the Model-View-Controller (MVC) design workflow:

- *Models* represent the data.
- *Views* visually represent the data model.
- *Controllers* route user requests and subsequent server responses

MVC provides a means of splitting the back-end business logic from the front-end views, and it's used to simplify the development process by allowing you to develop your application in phases or chunks.

Next we need to modify the default controller, *default.py*, so click the "edit" button next to the name of the file. Now we're in the **web2py IDE**. Replace the `index()` function with the following code:

```
def index():
    return dict(message="Hello from web2py!")
```

Save the file, then hit the Back button to return to the **Edit** page. Now let's update the view. Edit the *default/index.html* file, replacing the existing code with:

```
<html>
<head>
    <title>Hello App!</title>
</head>
<body>
    <br/>
    <h1>{{=message}}</h1>
</body>
</html>
```

Save the file, then return to the **Edit** page again. Click the "hello_world" title link at the top of the page. You should see the greeting, "Hello from web2py!" staring back at you.

Easy right?

So what happened?

The controller returned a dictionary with the key/value pair `{message="Hello from web2py"}`. Then, in the view, we defined how we wanted the greeting to be displayed by the browser. In other words, the functions in the controller return dictionaries, which are then converted into the outputs within the view when surrounded by `{}{...}{}` tags. The values from the dictionary are used as variables. In the beginning, you won't need to worry about the views since web2py has so many pre-made views already built in (again, defaults). So, you can focus solely on back-end development.

When a dictionary is returned, web2py looks to associate the dictionary with a view that matches the following format: `[controller_name]/[function_name].[extension]`. If no extension is specified, it defaults to `.html`, and if web2py cannot find the view, it defaults to using the `generic.html` view:

Views

[download layouts](#)

Edit	 _init__.py
Edit	 appadmin.html extends layout.html
Edit	 default/index.html
Edit	 default/user.html extends layout.html
Edit	 generic.html extends layout.html
Edit	 generic.ics
Edit	 generic.json
Edit	 generic.jsonp
Edit	 generic.load



For example, if we created this structure:

- Controller = `run.py`
- Function = `hello()`
- Extension `.html`

Then the url would be: `http://your_site/run/hello.html` .

In the `hello_world` app, since we used the `default.py` controller with the `index()` function, web2py looked to associate this view with `default/index.html`.

You can also easily render the view in different formats, like JSON or XML, by simply updating the extension:

- Go to http://localhost:8000/hello_world/default/index.html
- Change the extension to render the data in a different format:
 - **XML:** http://localhost:8000/hello_world/default/index.*xml*
 - **JSON:** http://localhost:8000/hello_world/default/index.*json*

Try this out. When done, hit CRTL-C from your terminal to stop the server.

Deploying on PythonAnywhere

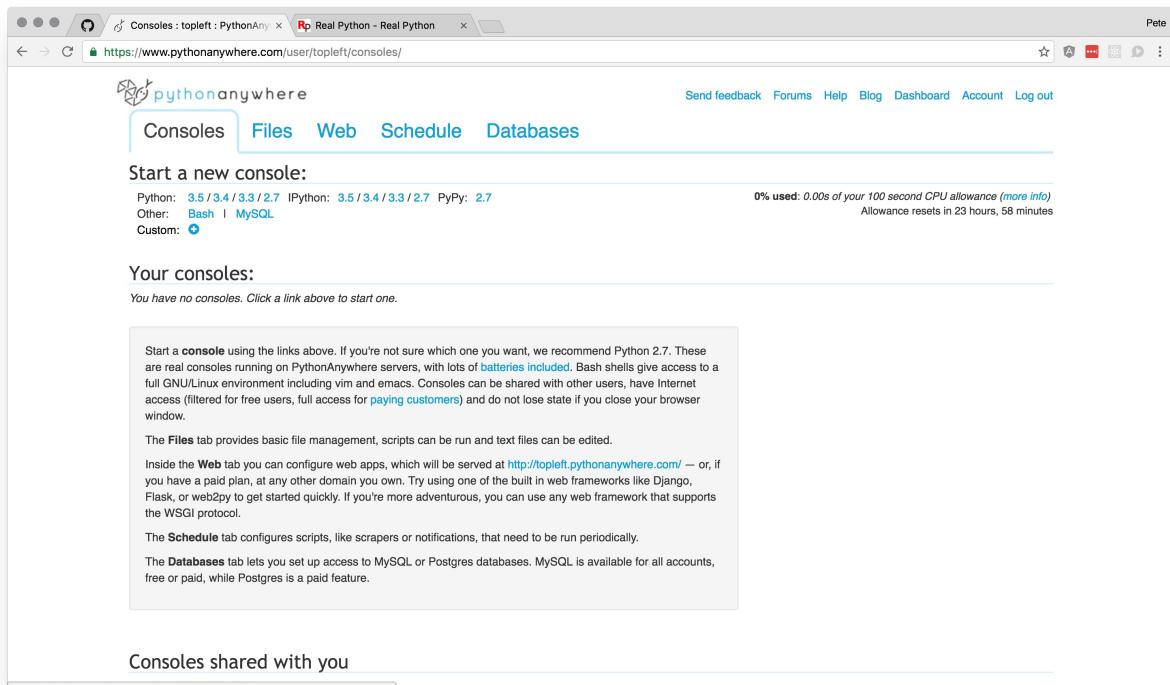
As its name suggests, [PythonAnywhere](#) is a Python-hosting platform that allows you to develop within your browser from [anywhere](#) in the world where there's an Internet connection.



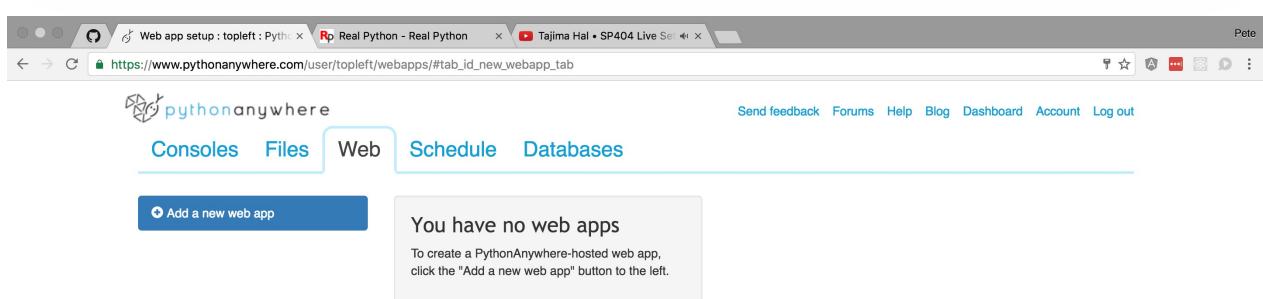
Simply invite a friend or colleague to join your session, and you have a collaborative environment for pair programming projects - or for getting help with a certain script you can't get working. It has a lot of other useful [features](#) as well, such as Drop-box integration and Python Shell access, among others.

Start by creating an account and login in. Once logged in, click "Web", then "Add a new web app". Your new PythonAnywhere app url will be

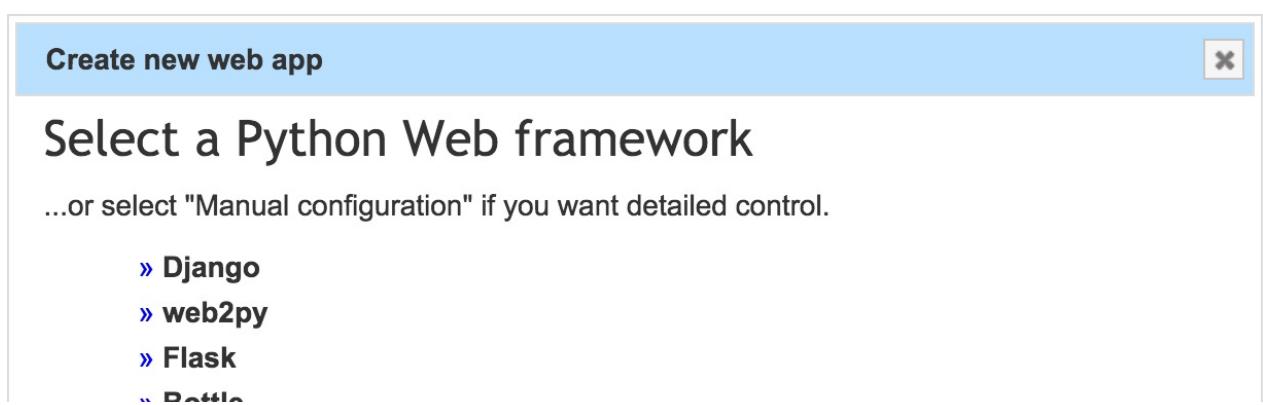
https://your_username.pythonanywhere.com. Click "Next" and then on this page click the button for web2py. "Next" again. Set an admin password and then click "Next" one last time to set up the web2py project.



The screenshot shows the PythonAnywhere 'Consoles' tab. At the top, there are links for 'Send feedback', 'Forums', 'Help', 'Blog', 'Dashboard', 'Account', and 'Log out'. Below that, there are tabs for 'Consoles', 'Files', 'Web', 'Schedule', and 'Databases', with 'Consoles' being the active tab. A sub-header 'Start a new console:' is followed by a table showing available Python versions: Python: 3.5 / 3.4 / 3.3 / 2.7, IPython: 3.5 / 3.4 / 3.3 / 2.7, and PyPy: 2.7. Other options include Bash, MySQL, and Custom. A note indicates 0% used of the 100-second CPU allowance, with a link to 'more info' and a note that the allowance resets in 23 hours, 58 minutes. Below this, a section titled 'Your consoles:' states 'You have no consoles. Click a link above to start one.' A large text box provides instructions for using consoles, mentioning batteries included, Bash shells, and the availability of vim and emacs. It also describes the 'Files' tab for basic file management, the 'Web' tab for configuring web apps, the 'Schedule' tab for periodic tasks, and the 'Databases' tab for MySQL or Postgres access. The MySQL option is noted as available for all accounts, while Postgres is a paid feature.



The screenshot shows the PythonAnywhere 'Web app setup' tab. The URL in the address bar is https://www.pythonanywhere.com/user/topleft/webapps/. The top navigation bar is identical to the 'Consoles' tab. The 'Web' tab is active. A blue button on the left says 'Add a new web app'. A central box displays the message 'You have no web apps' with the sub-instruction: 'To create a PythonAnywhere-hosted web app, click the "Add a new web app" button to the left.'



The screenshot shows a 'Create new web app' dialog box. The title bar says 'Create new web app' and has a close button. The main content area is titled 'Select a Python Web framework' in large, bold text. Below it, a sub-instruction reads '...or select "Manual configuration" if you want detailed control.' A list of frameworks is provided: '» Django', '» web2py', '» Flask', and '» Paste'. The 'Django' item is the first in the list.

» [DotEnv](#)

» [Manual configuration \(including virtualenvs\)](#)

What other frameworks should we have here? Send us some feedback using the link at the top of the page!

[Cancel](#)

[« Back](#)

[Next »](#)

Create new web app



Quickstart new web2py project

This will start web2py on your domain. If you already have a web2py app you want to use, you'll be able to use the web2py admin to install it.



Directory

/home/topleft/web2py/



Enter the path for a new directory to contain the web2py code

Admin Password

.....



Confirm Password

.....



Choose your password for the web2py admin site

(NB: this may take a couple of minutes)

[Cancel](#)

[« Back](#)

[Next »](#)

Navigate to https://your_username.pythonanywhere.com. (Note the https in the url.)

Look familiar? It better. Open the Admin Interface just like before and you can now start building your application.

NOTE: If you really wanted, you could develop your entire app on PythonAnywhere. Cool, right?

Back on your local version of web2py.

Before we deploy, we need to get the pydal package (needed to run web2py) into our `web2py/site-packages` directory. Run this:

```
$ pyenv exec pip install pydal -t ./site-packages/
```

Return to the [admin page](#), click the "Manage" drop down button next to your `hello_world` project, then select "Pack All" to save the `w2p-package` to your computer.

Installed applications

Application	Actions
admin (currently running)	Manage
examples	Manage Disable
hello_world	Manage Disable
welcome	Manage

Manage dropdown for welcome:

- Edit
- About
- Errors
- Clean
- Pack all
- Pack custom
- Compile (skip failed views)
- Compile (all or nothing)
- Uninstall

Once downloaded return to the Admin Interface on PythonAnywhere. To create your app, go to the "Upload and install packed application" section on the right side of the page, give your app a name ("hello_World"), and finally upload the `w2p-file` you saved to your computer earlier. Click install.

Navigate to your app's homepage:

https://your_username.pythonanywhere.com/hello_world/default/index

Congrats! You just deployed your first web2py app!

NOTE: If you get the following error after deploying - type

```
'exceptions ImportError' > No module named janrain_account
```

 - then you probably need to update web2py in web2py's admin page on PythonAnywhere.

Homework

- Python Anywhere is considered a Platform as a Service (PaaS). Find out exactly what that means. What's the difference between a PaaS hosting solution vs. shared hosting?
- Learn more about other Python PaaS options:
<http://www.slideshare.net/appsembler/pycon-talk-deploy-python-apps-in-5-min-with-a-paas>

seconds2minutes App

Let's create a new app that converts, well, seconds to minutes. Activate your virtual environment, start the server, set a password, enter the Admin Interface, and create a new app called "seconds2minutes".

NOTE: We'll be developing this locally. However, feel free to try developing it directly on PythonAnywhere. Or: Do both.

In this example, the controller will define two functions. The first function, `index()`, will return a form to `index.html`, which will then be displayed for users to enter the number of seconds they want converted over to minutes. Meanwhile, the second function, `convert()`, will take the number of seconds, converting them to the number of minutes. Both the variables are then passed to the `convert.html` view.

Replace the code in `default.py` with:

```
def index():
    form=FORM('# of seconds: ',
              INPUT(_name='seconds', requires=IS_NOT_EMPTY()),
              INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('convert', args=form.vars.seconds))
    return dict(form=form)

def convert():
    seconds = request.args(0,cast=int)
    return dict(seconds=seconds, minutes=seconds/60, new_seconds=seconds%60)
```

Edit the `default/index.html` view, replacing the default code with:

```
<center>
<h1>seconds2minutes</h1>
<h3>Please enter the number of seconds you would like converted to minutes.</h3>
<p>{{=form}}</p>
</center>
```

Create a new view called `default/convert.html`, replacing the default code with:

```

<center>
<h1>seconds2minutes</h1>
<p>{{=seconds}} seconds is {{=minutes}} minutes and {{=new_seconds}} seconds.</p>
<br/>
<p><a href="/seconds2minutes/default/index">Try again?</a></p>
</center>

```

Check out the live app. Test it out.

When we created the form, as long as a value is entered, we will be redirected to *convert.html*. Try entering no value, as well as a string or float. Currently, the only validation we have is that the form doesn't show up blank. Let's alter the code to add additional validators.

Change the form validator to `requires=IS_INT_IN_RANGE(0,1000000)` . The new *index()* function looks like this:

```

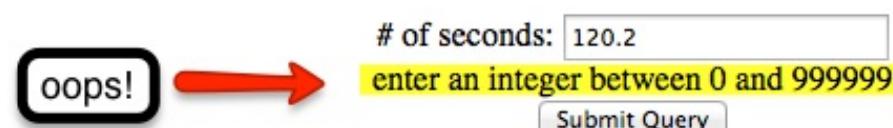
def index():
    form=FORM('# of seconds: ',
              INPUT(_type='integer', _name='seconds', requires=IS_INT_IN_RANGE(0,1000000
              )),
              INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('convert',args=form.vars.seconds))
    return dict(form=form)

```

Test it out. You should get an error, unless you enter an integer between 0 and 999,999:

seconds2minutes

Please enter the number of seconds you would like converted to minutes.



oops!

of seconds: 120.2

enter an integer between 0 and 999999

Submit Query

Again, the `convert()` function takes the seconds and then runs the basic expressions to convert the seconds to minutes. These variables are then passed to the dictionary and are used in the *convert.html* view.

Homework

- Deploy this app on PythonAnywhere.
- Want some fun? Try writing the same app with Flask.

Interlude: APIs

This chapter focuses on client-side programming. Clients are simply web browsers that access documents from other servers. Web server programming, on the other hand - covered in the next chapter - deals with, well, web servers. Put simply, when you browse the Internet, your web client (i.e., browser) sends a (GET) request to a remote server, which responds back to the web client with the requested information (usually HTML, CSS, and JavaScript).

In this chapter, we will navigate the Internet using Python programs to:

- gather data
- access and consume web services
- scrape web pages
- interact with web pages

This chapter assumes that you have some familiarity with HTML (HyperText Markup Language), the primary language of the Internet. If you need a quick brush up, the first 17 chapters of W3schools.com's [Basic HTML Tutorial](#) will get you up to speed quickly. They shouldn't take more than an hour to review. **Or simply review the chapter in this course on HTML and CSS.**

Make sure that at a minimum, you understand the basic elements of an HTML page such as the `<head>` and `<body>` as well as various HTML tags like `<a>` , `<div>` , `<p>` , `<h1>` , `` , `<center>` , and `
` .

Finally, to fully explore this topic, client-side programming, you can gain an understanding of everything from sockets to various web protocols and become a real expert on how the Internet works. We will not be going anywhere near that in-depth in this course. Our focus, rather, will be on the higher-level concepts, which are practical in nature and which can be immediately useful to a web development project. We will provide the required concepts, but it's more important to concern yourself with the actual programs and coding.

Homework

- Do you know the difference between the Internet and the Web? Did you know that there is a difference?

First, the Internet is a gigantic system of decentralized, yet interconnected computers that communicate with one another via protocols. Meanwhile, the web is what you see when you view web pages. In a sense, it's just a layer that rests on top of the Internet.

The Web is what most people think of as the Internet, which, now you know is actually incorrect.

- Read more about the differences between the Internet and the Web via Google. Look up any terminology that you have questions about, some of which we will be covering in this Chapter.

Retrieving Web Pages

The `requests` library is used for interacting with web pages. For straightforward situations, `requests` is very easy to use. You simply use the `'get()'` function, specify the URL you wish to access, and the web page is retrieved. From there, you can crawl or navigate through the web site or extract specific information.

Let's start with a basic example. But first, create a new folder within the "realpython" directory called "client-side". Don't forget to create and activate a new virtual environment.

Install:

```
$ pip install requests==2.11.1
```

Code:

```
# Retrieving a web page

import requests

# retrieve the web page
r = requests.get("http://www.python.org/")

print(r.content)
```

As long as you are connected to the Internet this script will pull the HTML source code from the Python Software Foundation's website and output it to the screen. It's a mess, right? Can you recognize the header (`<head> </head>`)? How about some of the other basic HTML tags?

Let's look at an easier way to view the full HTML output:

```
# Downloading a web page

import requests

r = requests.get("http://www.python.org/")

# write the content to test_request.html
with open("test_requests.html", "wb") as code:
    code.write(r.content)
```

Save the file as *clientb.py* and run it. If you don't see an error, you can assume that it ran correctly. Open the new file, *test_requests.html* in your text editor. Now it's much easier to examine the actual HTML. Go through the file and find tags that you recognize. Google any tags that you don't. This will help you later when we start web scraping.

NOTE: "wb" stands for write binary, which downloads the raw bytes of the file. In other words, the file is downloaded in its exact format.

Did you notice the `get()` function in those last two programs? Computers talk to one another via HTTP methods. The two methods you will use the most are GET and POST. When you view a web page, your browser uses GET to fetch that information. When you submit a form online, your browser will POST information to a web server. Make them your new best friends. *More on this later.*

Let's look at an example of a POST request:

```
# Submitting to a web form

import requests

url = 'http://httpbin.org/post'
data = {'fname': 'Michael', 'lname': 'Herman'}

# submit post request
r = requests.post(url, data=data)

# display the response to screen
print(r)
```

Output:

```
<Response [200]>
```

We created a dictionary with the field names as the keys `fname` and `lname`, associated with values `Michael` and `Herman` respectively.

`requests.post` initiates the POST request. In this example, you used the website <http://httpbin.org>, which is specifically designed to test HTTP requests, and received a response back in the form of a code, called a status code.

Common status codes:

- **200 OK**
- 300 Multiple Choices
- 301 Moved Permanently
- **302 Found**
- 304 Not Modified
- 307 Temporary Redirect
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- **404 Not Found**
- 410 Gone
- **500 Internal Server Error**
- 501 Not Implemented
- 503 Service Unavailable
- 550 Permission denied

There are actually many [more](#) status codes that mean various things, depending on the situation. However, the codes in **bold** above are the most common.

You should have received a "200" response to the POST request above.

Modify the script to see the entire response by appending `.content` to the end of the `print` statement:

```
print(r.content)
```

Save this as a new file called `clientd.py`. Run the file.

You should see the data you sent within the response:

```
"form": {  
  "lname": "Herman",  
  "fname": "Michael"  
},
```

Web Services Defined

Web services can be a difficult subject to grasp. Take your time with this. Learn the concepts in order to understand the exercises. Doing so will make your life as a web developer much easier. In order to understand web services, you first need to understand APIs.

APIs

An [API](#) (Application Programming Interfaces) is a type of protocol used as a point of interaction between two independent applications with the goal of exchanging data. Protocols define the type of information that can be exchanged. In other words, APIs provide a set of instructions, or rules, for applications to follow while accessing other applications.

One example that you've already seen is the SQLite API, which defines the SELECT, INSERT, UPDATE, and DELETE requests. The SQLite API allows the end user to perform certain tasks, which, in general, are limited to those four functions.

HTTP APIs

HTTP APIs, also called [web services](#), are simply APIs made available over the Internet, used for reading (GET) and writing (POST) data as well as updating existing data (UPDATE) and deleting data (DELETE).

CRUD	HTTP	SQL
CREATE	POST	INSERT
READ	GET	SELECT
UPDATE	PUT	UPDATE
DELETE	DELETE	DELETE

So, the SELECT command, for example, is equivalent to the GET HTTP method, which corresponds to the Read CRUD Operation.

Anytime you browse the Internet, you are constantly sending HTTP requests. For example, Real Python reads (GETs) data from Facebook to show how many people have "liked" a specific blog article. Then, if you "like" an article, data is sent (POST) to Facebook (if you allow it, of course), showing that you like that article. Without web services, these interactions between two independent applications would be impossible.

Let's look at an example in Chrome Developer Tools. Start by opening Chrome and navigating to <https://realpython.com>. Right click anywhere on the screen and within the window, click "Inspect Element". You are now looking at the main Developer Tools pane:

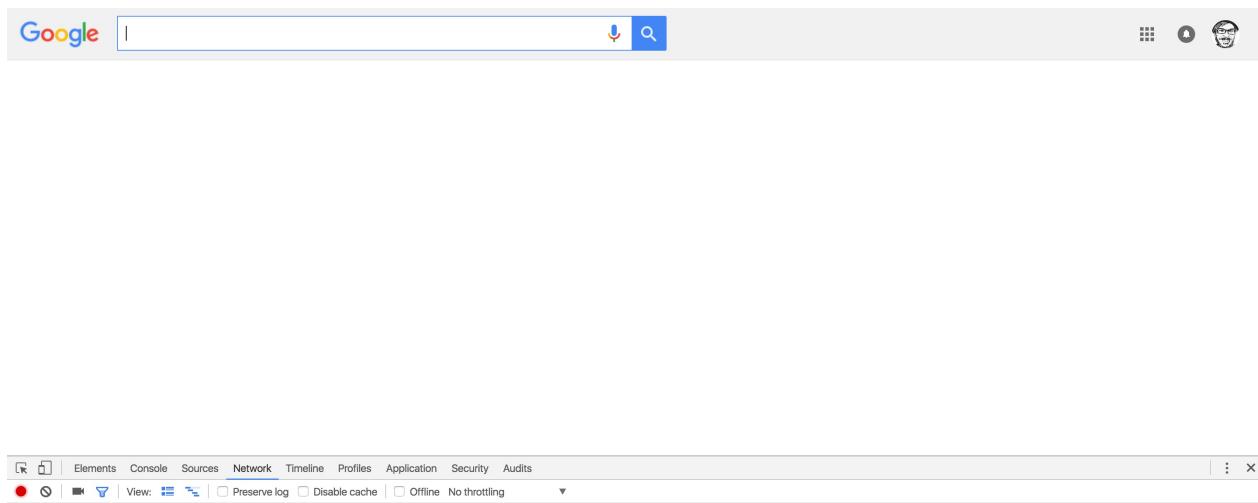
The screenshot shows the Real Python website with the title "Python Programming By Example". Below the title, there is a promotional message: "Real Python teaches programming and web development through hands-on, interesting examples that are useful and fun! Join the thousands who have already benefited from these unique Python courses and download your copy today. **Get three courses, with over 1,300 pages of content - packed with exercises, sample files, assignments, and bonus videos for only \$60!**". A green button says "Download Now > \$60". Below the button, it says "Or, view the other pricing options." To the right of the text, there is a logo for "REAL python™" with a stylized Python icon. At the bottom of the screenshot, the Chrome Developer Tools Network panel is visible, showing a list of network requests and responses. The panel has tabs for "Elements", "Console", "Sources", "Network", "Timeline", "Profiles", "Application", "Security", and "Audits". The "Network" tab is selected, and the list of requests includes "html", "video", and "body.hide-overflow.no-sidebar". The "Console" tab shows the command "document.querySelector('body').style.overflow = 'hidden';". The "Sources" tab shows the source code for "body.hide-overflow.no-sidebar". The "Network" tab shows the network activity with requests to "assets/bootstrap/dist/js/bootstrap.min.js" and "assets/bootstrap/dist/css/bootstrap.min.css". The "Timeline" tab shows the timeline of events. The "Profiles" tab shows the CPU and Memory usage. The "Application" tab shows the application state. The "Security" tab shows the security status. The "Audits" tab shows the audit results.

Click the Network panel. Right click on the column headers and be sure 'method' is selected. If you are not sure what I mean, click around, explore this interface.

NOTE: Being very familiar with the Chrome Dev Tools console will come in handy in the future. If you have trouble replicating the views in the following pictures, don't be afraid to click around and try things out. Google what you need if you still can't get them to look like the following images.

Now navigate in your browser to a site you need to login to view. Watch the activity in your console. Do you see the GET requests? You basically sent a request asking the server to display information for you.

Web Services Defined



Now log in to the site. Enter your login credentials. Pay close attention to the console. Did you see the POST request? Basically, you sent a POST request with your login credentials to the server.

Open Tree

Tree Logout

Repo

Search all GitHub repos by user:

Repo Owner

Enter GitHub username...

Repo Name

Enter GitHub repo name...

Submit

Recent

DOMINANT LANGUAGE: FILES: REPO STARS:

Elements Console Sources Network Timeline Profiles Application Security Audits

Request URL: https://lh3.googleusercontent.com/-tFPmxJ6Dv74/AAAAAAAIA/AAAAAAA/AEM0YSB5USGCUdaR2Pt4Gf060wVCFnPaBQ/s64-c-mo/photo.jpg

Request Method: GET

Status Code: 200 (from disk cache)

Remote Address: [2607:f8b0:400f:802::2001]:443

Response Headers

access-control-allow-origin: *

access-control-expose-headers: Content-Length

age: 862

alt-svc: quic=":443"; ma=2592000; vs="36,35,34"

cache-control: public, max-age=86400, no-transform

35 requests | 122 KB transferred |

bootstrap.min.css

git-tree.herokuapp.com

repos

ng-inspector.js

36 requests | 20.2 KB transferred |

web2py.app.admin (4).w2p

web2py.app.hello_world.w2p

Show All

Check out some other web pages. Try logging in to some of your favorite sites to see more POST requests. Perhaps POST a comment on a blog or message forum.

Applications can access APIs either directly, through the API itself, or indirectly, through a client library. The best means of access depends on a number of factors. Access through client libraries can be easier, especially for beginners, as the code to directly access the API has already been written. However, you still have to learn how the client library works and integrate the library's code base into your overall code. Also, if you do not first take the time to learn how the client library works, it can be difficult to debug or troubleshoot. Direct access provides greater control, but beginners may encounter more problems understanding and interpreting the rules of the specific API.

We will be looking at both methods.

NOTE: Not all web services rely on HTTP requests to govern the allowed interaction. Only RESTful APIs use POST, GET, PUT, and DELETE. This confuses a lot of developers. Just remember that RESTful APIs are just one type of web service. **Also, it's not really important to understand the abstract principles of RESTful design. Simply being able to recognize at a high-level what it is and the associated four HTTP methods is sufficient.**

Summary

In summary, APIs:

- Facilitate the exchange of information
- Speak a common language
- Can be accessed either directly or indirectly through client libraries

Although web services have brought much order to the Internet, the services themselves are still fairly chaotic. There are no standards besides a few high-level best practices (RESTful APIs) associated with HTTP requests. Documentation is a big problem too, as it is left to the individual developers to document how their web services work. If you start working more with web services, which we encourage you to do so, you will begin to see not only just how different each and every API is but also how terribly documented many of them are.

If you'd like more information on Web APIs, check out the [How to use APIs with Python](#) crash course from Codecademy.

Fortunately, data exchanged via web services is standardized in text-based formats and thus, are both human and machine-readable. Two popular formats used today are XML and JSON, which we will address shortly.

Before moving on, let's look at a fun example. <http://placekitten.com> is an API that returns a picture of a kitten given a width and height - <http://placekitten.com/WIDTH/HEIGHT>. You can test it out right in your browser; just navigate to these URLs:

- <http://placekitten.com/200/300>
- <http://placekitten.com/300/450>
- <http://placekitten.com/700/600>

Homework

- Read [this](#) article providing a high-level overview of standards associated with RESTful APIs.
- If you're still struggling with understanding what Web Services as well as RESTful APIs, check out [Teach a Dog to REST](#). After about 8:30 minutes it starts to get pretty technical, but before that it's pretty accessible to everyone.

Working with XML

XML (eXtensible Markup Language) is a highly structured language, designed specifically for transferring information. The rigid structure makes it perfect for reading and interpreting the data (called parsing) found within an XML file. It's both human and machine-readable.

NOTE: Although, JSON continues to push XML out of the picture in terms of web services, XML is still widely used and can be easier to parse.

With that, let's look at an example of an XML file:

```
<?xml version="1.0"?>
<CARS>
  <CAR>
    <MAKE>Ford</MAKE>
    <MODEL>Focus</MODEL>
    <COST>15000</COST>
  </CAR>
  <CAR>
    <MAKE>Honda</MAKE>
    <MODEL>Civic</MODEL>
    <COST>20000</COST>
  </CAR>
  <CAR>
    <MAKE>Toyota</MAKE>
    <MODEL>Camry</MODEL>
    <COST>25000</COST>
  </CAR>
  <CAR>
    <MAKE>Honda</MAKE>
    <MODEL>Accord</MODEL>
    <COST>22000</COST>
  </CAR>
</CARS>
```

There's a declaration at the top, and the data is surrounded by opening and closing tags. One useful thing to remember is that the purpose of XML is much different than HTML. While HTML is used for displaying data, XML is used for transferring data. In itself, an XML document is purposeless until it is read, understood, and parsed by an application. It's about what you *do* with the data that matters.

With that in mind, let's build a quick parser. There are quite a few libraries you can use to read and parse XML files. One of the easiest libraries to work with is the [ElementTree](#) library, which is part of Python's standard library.

Use the `cars.xml` file found in the exercises repo for this example.

Code:

```
# XML Parsing 1

from xml.etree import ElementTree as et

# parses the file
doc = et.parse("cars.xml")

# outputs the first MODEL in the file
print(doc.find("CAR/MODEL").text)
```

Output:

Focus

In this program we read and parsed the file using the `find` function and then outputted the data between the first `<MODEL>` `</MODEL>` tags. These tags are called element nodes, and are organized in a tree-like structure and further classified into parent and child relationships.

In the example above, the parent is `<CARS>`, and the child elements are `<CAR>`, `<MAKE>`, `<MODEL>`, and `<cost>`. The `find` function begins looking for elements that are children of the parent node, which is why we started with the first child when we outputted the data, rather than the parent element:

```
print(doc.find("CAR/MODEL").text)
```

The above line is equivalent to:

```
print(doc.find("CAR[1]/MODEL").text)
```

See what happens when you change the code in the program to:

```
print(doc.find("CAR[2]/MODEL").text)
```

The output should be:

```
Civic
```

See how easy that was. That's why XML is both machine *and* human readable. Let's take it a step further and add a loop to extract all the data:

```
# XML Parsing 2

from xml.etree import ElementTree as et

doc = et.parse("cars.xml")

# outputs the make, model and cost of each car to the screen
for element in doc.findall("CAR"):
    print(element.find("MAKE").text + " " +
          element.find("MODEL").text +
          ", $" + element.find("COST").text)
```

You should get the following results:

```
Ford Focus, $15000
Honda Civic, $20000
Toyota Camry, $25000
Honda Accord, $22000
```

This program follows the same logic as the previous one, but we just added a `for` loop to iterate through the XML file, pulling all the data and then outputting it.

Finally, in this last example, we will use a GET request to access XML found on the web:

```
# XML Parsing 3

from xml.etree import ElementTree as et
import requests

# retrieve an xml document from a web server
xml = requests.get("http://www.w3schools.com/xml/cd_catalog.xml")

with open("test.xml", "wb") as code:
    code.write(xml.content)

doc = et.parse("test.xml")

# outputs the album, artist and year of each CD to the screen
for element in doc.findall("CD"):
    print("Album: ", element.find("TITLE").text)
    print("Artist: ", element.find("ARTIST").text)
    print("Year: ", element.find("YEAR").text, "\n")
```

Again, this program follows the same logic. You just added an additional step by importing the requests library and downloading the XML file before reading and parsing the XML.

Working with JSON

JSON (JavaScript Object Notation) is a lightweight format used for transferring data. Like XML, it's both human and machine readable, which makes it easy to generate and parse, and it's used by thousands of web services. Its syntax differs from XML though, which many developers prefer because it's faster and takes up less memory. Because of this, JSON is becoming the format of choice for web services. It's derived from Javascript and, as you will soon see, resembles a Python dictionary.

Let's look at a quick example:

```
{  
  "CARS": [  
    {  
      "MAKE": "Ford",  
      "MODEL": "Focus",  
      "COST": "15000"  
    },  
    {  
      "MAKE": "Honda",  
      "MODEL": "Civic",  
      "COST": "20000"  
    },  
    {  
      "MAKE": "Toyota",  
      "MODEL": "Camry",  
      "COST": "25000"  
    },  
    {  
      "MAKE": "Honda",  
      "MODEL": "Accord",  
      "COST": "22000"  
    }  
  ]  
}
```

Although the data looks very similar to XML, there are many noticeable differences. There's less code, no start or end tags, and it's easier to read. Also, because JSON operates much like a Python dictionary, it is very easy to work with with Python.

Basic Syntactical rules:

1. Data is found in key/value pairs (i.e., `"MAKE": "FORD"`).
2. Data is separated by commas.

3. The curly brackets contain dictionaries, while the square brackets hold lists.

JSON decoding is the act of taking a JSON file, parsing it, and turning it into something usable.

Code:

```
# JSON Parsing 1

import json

# decodes the json file
output = json.load(open('cars.json'))

# display output to screen
print(output)
```

Output:

```
[{u'CAR': [{u'MAKE': u'Ford', u'COST': u'15000', u'MODEL': u'Focus'}, {u'MAKE': u'Honda', u'COST': u'20000', u'MODEL': u'Civic'}, {u'MAKE': u'Toyota', u'COST': u'25000', u'MODEL': u'Camry'}, {u'MAKE': u'Honda', u'COST': u'22000', u'MODEL': u'Accord'}]]
```

You see we have four dictionaries inside a list, enclosed within another dictionary, which is finally enclosed within another list. *Repeat that to yourself a few times*. Can you see that in the output?

If you're having a hard time, try changing the print statement to:

```
print(json.dumps(output, indent=4, sort_keys=True))
```

Your output should now look like this:

```
[  
  {  
    "CAR": [  
      {  
        "COST": "15000",  
        "MAKE": "Ford",  
        "MODEL": "Focus"  
      },  
      {  
        "COST": "20000",  
        "MAKE": "Honda",  
        "MODEL": "Civic"  
      },  
      {  
        "COST": "25000",  
        "MAKE": "Toyota",  
        "MODEL": "Camry"  
      },  
      {  
        "COST": "22000",  
        "MAKE": "Honda",  
        "MODEL": "Accord"  
      }  
    ]  
  }  
]
```

Much easier to read, right?

If you want to print just the value "Focus" of the "MODEL" key within the first dictionary in the list, you could run the following code:

```
# JSON Parsing 2  
  
import json  
  
# decodes the json file  
output = json.load(open('cars.json'))  
  
# display output to screen  
print output[0]["CAR"][0]["MODEL"]
```

Let's look at the `print` statement in detail:

1. `[0]["CAR"]` - indicates that we want to find the first car dictionary. Since there is only one, there can only be one value - `0`.
2. `[0]["MODEL"]` - indicates that we want to find the first instance of the `model` key,

and then extract the value associated with that key. If we changed the number to `1`, it would find the second instance of `model` and return the associated value: `Civic`.

Finally, let's look at how to POST JSON to an API:

```
# POST JSON Payload

import json
import requests

url = "http://httpbin.org/post"
payload = {"colors": [
    {"color": "red", "hex": "#f00"}, 
    {"color": "green", "hex": "#0f0"}, 
    {"color": "blue", "hex": "#00f"}, 
    {"color": "cyan", "hex": "#0ff"}, 
    {"color": "magenta", "hex": "#f0f"}, 
    {"color": "yellow", "hex": "#ff0"}, 
    {"color": "black", "hex": "#000"}
]}
headers = {"content-type": "application/json"}

# post data to a web server
response = requests.post(url, data=json.dumps(payload), headers=headers)

# output response to screen
print(response.status_code)
```

Output:

```
200 OK
```

In some cases you will have to send data (a payload) to be interpreted by a remote server to perform some action on your behalf. For example, you could send a JSON Payload to Twitter with a number Tweets to be posted.

Some tech companies require you to send a JSON payload with your name, telephone number, email address, and a link to your online resume to apply to ensure you know the basics of sending a POST request. If you ever run into a similar situation, make sure to test the Payload before actually sending it to the real URL. You can use sites like [JSON Test](#) for testing purposes.

Working with Web Services

Now that you've seen how to communicate with web services (via HTTP methods) and how to handle the resulting data (either XML or JSON), let's look at some more robust examples.

Twitter

Like Google, Twitter provides a very open API. We use the Twitter API extensively for pulling in tweets on specific topics, which we then parse and extract into a CSV file for analysis. One of the best client libraries to use with the Twitter API is [Tweepy](#).

Install:

```
$ pip install tweepy
```

Before we look at an example, you need to obtain access codes. Navigate to <https://dev.twitter.com/apps>. Click the button to create a new application. Enter dummy data. Then register.

Create an application

Application Details

Name *
 
Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Developer Agreement

Yes, I have read and agree to the [Twitter Developer Agreement](#).

Once complete, you will be taken to your application where you will need the following access codes:

- consumer_key
- consumer_secret
- access_token
- access_secret

Now, let's look at an example:

```
# Twitter Web Services

import tweepy

consumer_key      = "<get_your_own>"
consumer_secret  = "<get_your_own>"
access_token      = "<get_your_own>"
access_secret     = "<get_your_own>"

auth = tweepy.auth.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_secret)
api = tweepy.API(auth)

tweets = api.search(q='#python')

# display results to screen
for t in tweets:
    print(t.created_at, t.text, "\n")
```

NOTE: Add your keys and tokens in the above code before running.

WARNING: Make sure you change your keys and token variables back to "
<get_your_own>" before you PUSH to Github. You do not want anyone else but you to get a hold of those keys and make requests on your behalf.

If done correctly, this should output the tweets and the dates and times they were created:

```
2016-11-12 23:18:00 Try/Except in Python: How do you properly ignore Exceptions? #
python #exception #exception-handling #try-except https://t.co/3nQaPrSbsN

2016-11-12 23:17:43 早起きしたので、ブログに記事を追加しました!

今回の記事は#wxPython でメニューバーを作成する方法です。

結構大変なんですよね('Δ')
```

```
https://t.co/5SEDtu3W0s

#python https://t.co/ToZkwyTYc0

2016-11-12 23:17:17 RT @msdevUK: A tidy collection of free #NodeJS #JS #AngularJS
#Git #Python programming books for developers: https://t.co/upNUoUeyIt #Dev h...

2016-11-12 23:15:03 It's like Sesame Street for the STEM set https://t.co/kI00n5U5
3P #edtech #edchat #linux #python #puppets #firefox #html5 #blender #lwks #diy

2016-11-12 23:15:00 Recruit tech talent for #Java, #Ruby, #Python, #Scala & #P
HP #Devops #Mobile #Apps #BigData #Hadoop & more. Visit https://t.co/zjWuBvQSx
Q

2016-11-12 23:12:57 RT @KirkDBorne: Coming soon from @jakevdp "#Python #DataScienc
e Handbook: Essential Tools for Working with Data" https://t.co/oXfxXWoqhx #M...

2016-11-12 23:12:56 RT @codetoday_: In other news...
We come to you to deliver #coding courses in #Python for your children and their f
riends. #London
https:...

2016-11-12 23:10:17 RT @KirkDBorne: Implement K-Nearest Neighbor (KNN) algorithm u
sing #Python: https://t.co/lQsFZuwn8H #abdsc #BigData #DataScience #MachineLe...

2016-11-12 23:09:57 RT @PythonEggs: A Beginner-level Tutorial to Ray: A New Python
Framework https://t.co/tYqJDrDVX6 #python https://t.co/nR4BLYSCSv

2016-11-12 23:06:02 It was created to help you #python #ssh #Yii #SearchEngineOptim
ization #Joomla #ReactJS #googleanalytics #git... https://t.co/xYHaMcFU0h

2016-11-12 23:06:01 Python: import a file from a subdirectory #python #module #sub
directory #python-import https://t.co/pzBCL00Ckk

2016-11-12 23:03:53 RT @d3f0: Lautaro Pecile mostrando programación funcional en e
l #meetup #patagónico de #python en @FI_UNPSJB #madryn https://t.co/h7cRG0gFUC

2016-11-12 23:01:08 Gini Art 13/11/2016
#DigitalArt #Art #Gimp #python https://t.co/e5rSt0cCNn

2016-11-12 23:00:16 [bot]This is INSANE https://t.co/jRfnZWYxjW #python

2016-11-12 23:00:03 2016-11-13 00:00:03 #devops Comparte ficheros con https://t.co
/s4AzFm5SsZ #comparte #python #falconframework #knockoutjs
```

Essentially, the search results were returned to a list, and then we iterated through that list to extract the desired information.

How did we know we wanted the keys `created at` and `text`? Again, trial and error. Start with the [documentation](#), then turn to Google. You will soon become quite adept at knowing the types of reliable sources to use when trying to obtain information about an API.

Try this on your own. See what other information you can extract. Have fun!

WARNING: REMEMBER!!! Make sure you change your keys and token variables back to `"<get_your_own>"` before you PUSH to Github.

Google Directions API

With the Google Directions API, you can obtain directions between two points for a number of different modes of transportation. Start by looking at the documentation [here](#). In essence, the documentation is split in two. The first part (requests) describes the type of information you can obtain from the API, and the second part details how to obtain (responses)said information.

Let's get walking directions from Central Park to Times Square. Use the following URL to call (or invoke) the API:

<https://maps.googleapis.com/maps/api/directions/output?parameters>

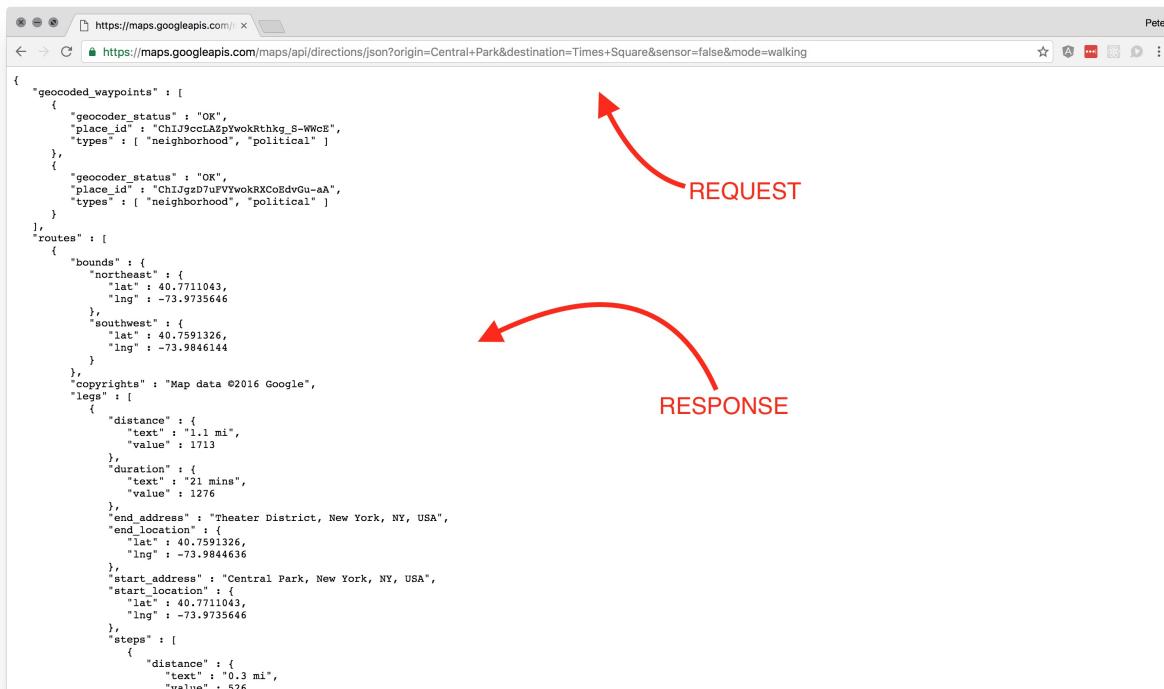
You then must specify the output. We'll use JSON since it's easier to work with. Also, you must specify some parameters. Notice in the documentation how some parameters are required while others are optional. Let's use these parameters:

- `origin=Central+Park`
- `destination=Times+Square`
- `sensor=false`
- `mode=walking`

Now you could simply append the output as well as the parameters to the end of the URL like so-

[https://maps.googleapis.com/maps/api/directions/json?
origin=Central+Park&destination=Times+Square&sensor=false&mode=walking](https://maps.googleapis.com/maps/api/directions/json?origin=Central+Park&destination=Times+Square&sensor=false&mode=walking)

-and then call the API directly from your browser:



However, there's a lot more information there than we need. Let's call the API directly from the Python Shell, and then extract the actual driving directions. In the python shell run these commands:

```

import requests, json

url = "https://maps.googleapis.com/maps/api/directions/json?origin=Central+Park&destination=Times+Square&sensor=false&mode=walking"

data = requests.get(url)
binary = data.content
output = json.loads(str(binary, 'utf-8'))
print(output['status'])

```

All right. Let's breakdown the nested `for` loops:

```

for route in output['routes']:
    for leg in route['legs']:
        for step in leg['steps']:
            print(step['html_instructions'])

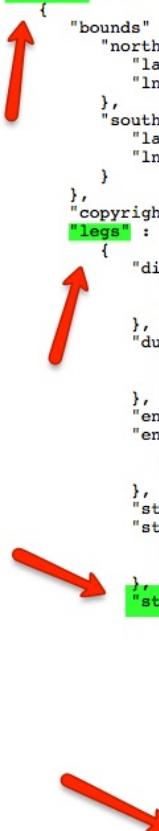
```

Working with Web Services

```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests, json
>>>
>>> url = "https://maps.googleapis.com/maps/api/directions/json?origin=Central+Park&destination=Times+Square&sensor=false&mode=walking"
>>>
>>> data = requests.get(url)
>>> binary = data.content
>>> output = json.loads(str(binary, 'utf-8'))
>>> print(output['status'])
OK
>>> for route in output['routes']:
...     for leg in route['legs']:
...         for step in leg['steps']:
...             print(step['html_instructions'])
...
Head <b>south</b> on <b>Center Drive</b>/<b>Frisbee Hill</b><div style="font-size:0.9em">Continue to follow Frisbee Hill</div>
Turn <b>left</b> toward <b>W 59th St</b>/<b>Central Park S</b>
Turn <b>right</b> toward <b>W 59th St</b>/<b>Central Park S</b>
Turn <b>right</b> onto <b>W 59th St</b>/<b>Central Park S</b>
Turn <b>left</b> onto <b>7th Ave</b>
Turn <b>left</b> onto <b>W 47th St</b>
>>> █
```

Compare the loops to the entire output. You can see that for each loop we're just moving in (or down) one level:

```
{
  "routes": [
    {
      "bounds": {
        "northeast": {
          "lat": 40.77361000000001,
          "lng": -73.97110000000001
        },
        "southwest": {
          "lat": 40.75890,
          "lng": -73.98529000000001
        }
      },
      "copyrights": "Map data ©2013 Google",
      "legs": [
        {
          "distance": {
            "text": "1.4 mi",
            "value": 2285
          },
          "duration": {
            "text": "28 mins",
            "value": 1676
          },
          "end_address": "Times Square, 1560 Broadway #800, New York, NY 10036, USA",
          "end_location": {
            "lat": 40.75890,
            "lng": -73.98529000000001
          },
          "start_address": "Central Park, 14 East 60th Street, New York, NY 10022, USA",
          "start_location": {
            "lat": 40.77361000000001,
            "lng": -73.97110000000001
          },
          "steps": [
            {
              "distance": {
                "text": "0.2 mi",
                "value": 328
              },
              "duration": {
                "text": "4 mins",
                "value": 241
              },
              "end_location": {
                "lat": 40.77079000000001,
                "lng": -73.97220
              },
              "html_instructions": "Head \u003cb\u003esouth\u003c/b\u003e on \u003cb\u003eThe Mall\u003c/b\u003e",
            }
          ]
        }
      ]
    }
  ]
}
```



So, if we wanted to print the `start_address` and `end_address`, we would just need two for loops:

```
for route in output['routes']:
    for leg in route['legs']:
        print(leg['start_address'])
        print(leg['end_address'])
```

Homework

- Using the Google Direction API, pull driving directions from San Francisco to Los Angeles in XML. Then extract the step-by-step directions.

My API Films

Before moving on to web scraping, let's look at an extended example of how to use web services to obtain information. In the last lesson we used client libraries to connect with APIs; in this lesson we'll establish a direct connection. You'll grab data - e.g., make a GET request - from the [My API Films](#), parse the relevant info, then upload the data to a SQLite database.

Start by navigating to the following URL in your browser -

<http://api.myapifilms.com/imdb.do>.

Whenever you start working with a new API, you always want to start with the documentation. Again, all APIs work differently because few universal standards or practices have been established. Fortunately, the *My API Films*' API is not only well documented - but also easy to read and follow.

In this example, let's grab a list of all movies currently playing in theaters, which we can grab from this endpoint:

<http://api.myapifilms.com/imdb.do#inTheatersExample> often Try it out in the browser.

Endpoints are the actual connection points for accessing the data. In other words, they are the specific URLs used for calling an API. Each endpoint is generally associated with a different type of data, which is why endpoints are often associated with groupings (often called resources) of data (e.g., movies playing in the theater, movies opening on a certain date, top rentals, and so on).

Did you notice how you need a token to make the actual API call?

The majority of web APIs (or web services) require users to go through some form of authentication in order to access their services. There are a number of different means of going through authentication.

In this particular case, we just need to request a token from [here](#). Once you obtain the token, DO NOT share it with anyone. You do not want someone else using that key to obtain information and possibly use it in an illegal or unethical manner.

Once you have your key, go ahead and test it out:

[http://api.myapifilms.com/imdb/inTheaters?
token=ADD_YOUR_TOKEN_HERE&format=json&language=en-us](http://api.myapifilms.com/imdb/inTheaters?token=ADD_YOUR_TOKEN_HERE&format=json&language=en-us)

Use the URL from above and replace "ADD_YOUR_TOKEN_HERE" with the generated token. Now test it in your browser. You should see a large JSON file full of data. If not, there may be a problem with your token. **Make sure you copied and pasted the entire token and appended it correctly to the URL.**

Now comes the fun part: Building the program to actually GET the data, parse the relevant data, and then dump it into a database. We'll do this in iterations.

Let's start by writing a script to create the database and table:

```
# Create a SQLite3 database and table

import sqlite3

with sqlite3.connect("movies.db") as connection:
    c = connection.cursor()

    # create a table
    c.execute("""CREATE TABLE new_movies
                (title TEXT, year INT, votes text,
                release_date text, rating INT, metascore INT)""")
```

Save this as *create_db.py* and then run it:

```
$ python create_db.py
```

We need one more script now to pull the data and dump it directly to the database:

```
# GET data from My API Films, parse, and write to database

import json
import requests
import sqlite3

TOKEN = '<UPDATE ME!!!>'
url = requests.get('http://api.myapifilms.com/imdb/inTheaters?token={0}&format=json&language=en-us'.format(TOKEN))

# convert data from feed to binary
binary = url.content

# decode the json feed
output = json.loads(str(binary, "utf-8"))

# grab the list of movies
movies = output['data']['inTheaters']

with sqlite3.connect("movies.db") as connection:
    c = connection.cursor()

    # iterate through each movie and write to the database
    for movie in movies:
        all_movies = movie['movies']
        for meta in all_movies:
            if(meta['title']):
                c.execute("INSERT INTO new_movies VALUES(?, ?, ?, ?, ?, ?, ?)",
                          (meta["title"], meta["year"], meta["votes"],
                           meta["releaseDate"], meta["metascore"],
                           meta["rating"]))

    # retrieve data
    c.execute("SELECT * FROM new_movies ORDER BY title ASC")

    # fetchall() retrieves all records from the query
    rows = c.fetchall()

    # output the rows to the screen, row by row
    for r in rows:
        print(r[0], r[1], r[2], r[3], r[4], r[5])
```

Save this as `get_movies.py`.

Make sure you add your API token into the value for the variable `TOKEN`.

What happened?

We:

1. Grabbed the data from the endpoint URL with a GET request.
2. Converted the data to binary.
3. Decoded the JSON feed.
4. Used a `for` loop to write the data to the database.
5. Selected the data from the database and outputted it.

Nice, right?

Were you able to follow this code? Go through it a few more times. See if you can grab data from a different endpoint. *Practice!*

web2py: Sentiment Analysis

NOTE: Continue to use the same web2py instance as before for the apps in this chapter as well.

Sentiment Analysis

What is Sentiment Analysis?

Essentially, [sentiment analysis](#) measures the sentiment of something - a feeling rather than a fact. The aim is to break down natural language data, analyze each word, and then determine if the data, as a whole, is positive, negative, or neutral.

Twitter is a great resource for sourcing data for sentiment analysis. You could use the Twitter API to pull hundreds of thousands of tweets on topics such as Obama, abortion, gun control, etc. to get a sense of how Twitter users feel about a particular topic. Companies often use sentiment analysis to gain a deeper understanding about marketing campaigns, product lines, and the company itself.

NOTE: Sentiment analysis works best when it's conducted on a popular topic that people have strong opinions about.

In this example, we'll be using a [natural language classifier](#) to power a web application that allows you to enter data for analysis via an html form. The focus is not on the classifier but on the development of the application. For more information on how to develop your own classifier using Python, read [Twitter sentiment analysis using Python and NLTK](#).

Steps

Start by reading the API [documentation](#) for the natural language classifier we'll be using for our app. Are the docs clear? What questions do you have? Write them down. You should be able to answer them by the end of this lesson.

First, what the heck is [cURL](#)? For simplicity, cURL is a utility used for transferring data across numerous protocols. We will be using it to test HTTP requests.

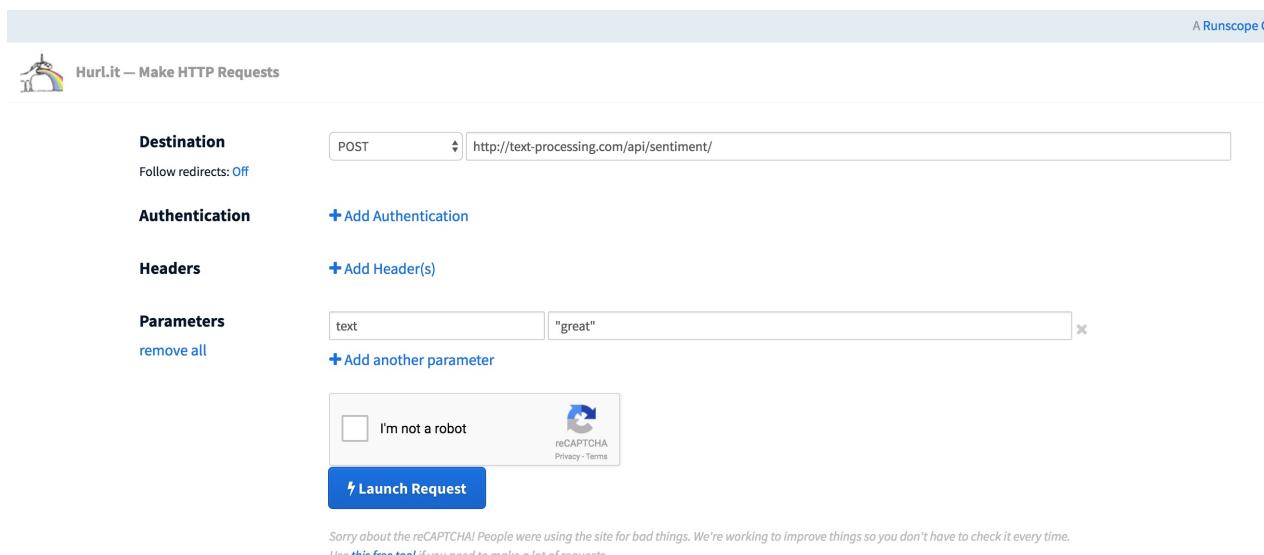
NOTE: Traditionally, you would access cURL from the bash terminal in Unix environments. Unfortunately, for Windows users, prior to [Windows 10](#), it does not come with the utility. Fortunately, there is an advanced command line tool called [Cygwin](#) available that provides a Unix-like terminal for Windows. Please follow the steps [here](#) to install. Make sure to add the cURL package when you get to the "Choosing Packages" step. Or scroll down to step 5 and use Hurl instead.

Test out the API in the terminal:

```
$ curl -d "text=great" http://text-processing.com/api/sentiment/  
  
{"probability": {"neg": 0.35968353095023886, "neutral": 0.29896828324578045, "pos": 0.64031646904976114}, "label": "pos"}  
  
$ curl -d "text=i hate apples" http://text-processing.com/api/sentiment/  
  
{"probability": {"neg": 0.65605365432549356, "neutral": 0.3611947857779943, "pos": 0.34394634567450649}, "label": "neg"}  
  
$ curl -d "text=i usually like ice cream but this place is terrible" http://text-processing.com/api/sentiment/  
  
{"probability": {"neg": 0.90030914036608489, "neutral": 0.010418429982506104, "pos": 0.099690859633915108}, "label": "neg"}  
  
$ curl -d "text=i really really like you, but today you just smell." http://text-processing.com/api/sentiment/  
  
{"probability": {"neg": 0.638029187699517, "neutral": 0.001701536649255885, "pos": 0.36197081230048306}, "label": "neg"}
```

So, you can see the natural language, the probability of the sentiment being positive, negative, or neutral, and then the final sentiment. Did you notice the last two text statements are more neutral than negative but were classified as negative? Why do you think that is? How can a computer analyze sarcasm?

You can also test the API on [Hurl](#):



The screenshot shows the Hurl.it interface for making HTTP requests. The request is a POST to <http://text-processing.com/api/sentiment/>. The 'text' parameter is set to "great". A reCAPTCHA verification box is present, asking "I'm not a robot" with a checkbox and a "Launch Request" button.

Steps:

- Enter the URL you wish to test
- Change the HTTP method to POST
- Add the parameter `text` and `"i greatly dislike perl"`
- Press send

Surprised at the results?

Requests

Alright, let's build the app. Before we begin, though, we will be using the `requests` library for initiating the POST request. The `cURL` command is equivalent to the following code:

```
import requests

url = 'http://text-processing.com/api/sentiment/'
data = {'text': 'great'}
r = requests.post(url, data=data)
print(r.content)
```

Go ahead and install the `requests` library. Wait. Didn't we already do that? *Remember: Since we're in a different virtual environment, we need to install it again.* Use `pip install requests`.

Test out the above code in your Shell:

```
(env) ➜ sentiment python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
>>> url = 'http://text-processing.com/api/sentiment/'
>>> data = {'text': 'great'}
>>> r = requests.post(url, data=data)
>>> print(r.content)
b'{"probability": {"neg": 0.30135019761690551, "neutral": 0.27119050546800266, "pos": 0.69864980238309449}, "label": "pos"}'
>>> █
```

Now, let's build the app for easily testing sentiment analysis. It's called "Pulse".

Pulse

You know the drill: fire up the server using `pyenv exec`, enter the Admin Interface, and create a new app called "Pulse".

Like the last app, the controller will define two functions, `index()` and `pulser()`. `index()`, will return a form to `index.html`, so users can enter the text for analysis. `pulser()`, meanwhile, handles the POST request to the API and outputs the results of the analysis to `pulser.html`.

Replace the code in the default controller with:

```
import requests

def index():
    form=FORM(
        TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
        INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('pulser',args=form.vars.pulse))
    return dict(form=form)

def pulser():
    text = request.args(0)
    text = text.split('_')
    text = ' '.join(text)
    url = 'http://text-processing.com/api/sentiment/'
    data = {'text': text}
    r = requests.post(url, data=data)
    return dict(text=text, r=r.content)
```

Now let's create some basic views.

`default/index.html`:

```
{{extend 'layout.html'}}
<center>
<br/>
<br/>
<h1>check a pulse</h1>
<h4>Just another Sentiment Analysis tool.</h4>
<br/>
<p>{{=form}}</p>
</center>
```

`default/pulser.html`:

```
 {{extend 'layout.html'}}  
<center>  
<p>{{=text}}</p>  
<br/>  
<p>{{=r}}</p>  
<br/>  
<p><a href="/pulse/default/index">Another Pulse?</a></p>  
</center>
```

Test this out. Compare the results to the results using either cURL or Hurl to make sure all is set up correctly. If you did everything right you should have received an error that the requests module is not installed. Kill the server, and then install requests from the terminal:

```
$ pyenv exec pip install requests
```

Test it again. Did it work this time?

Now, let's finish cleaning up *pulser.html*. We need to parse the JSON file. What do you think the end user wants to see? Do you think they care about the probabilities? Or just the end results? What about a graph? It all depends on your (intended) audience. Let's just parse out the end results for now.

Update *default.py*:

```
import requests
import json

def index():
    form=FORM(
        TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
        INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('pulser', args=form.vars.pulse))
    return dict(form=form)

def pulser():
    text = request.args(0)
    text = text.split('_')
    text = ' '.join(text)

    url = 'http://text-processing.com/api/sentiment/'
    data = {'text': text}

    r = requests.post(url, data=data)

    binary = r.content
    output = json.loads(binary)
    label = output["label"]

    return dict(text=text, label=label)
```

NOTE: Notice how we did not change the 'binary' data into a string using "utf-8" encoding like we did in previous chapters? Can you think of why? Remember that we are using Python 2.7 for all of our web2py projects, and there are some slight differences.

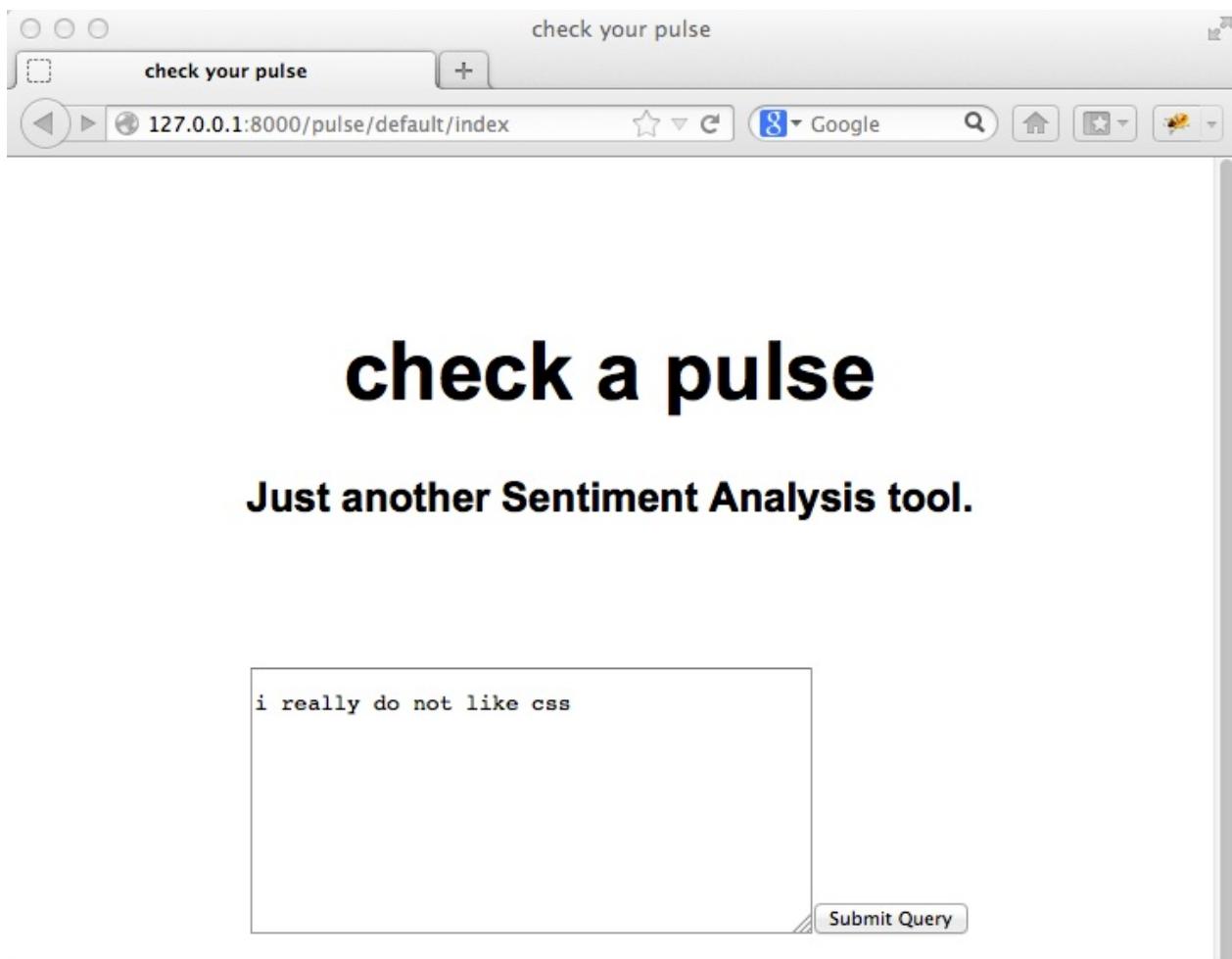
Update *default/pulser.html*:

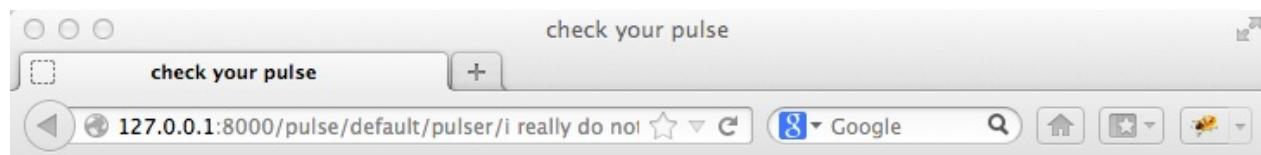
```
{{extend 'layout.html'}}
<center>
<br/>
<br/>
<h1>your pulse</h1>
<h4>{{=text}}</h4>
<p>is</p>
<h4>{{=label}}</h4>
<br/>
<p><a href="/pulse/default/index">Another Pulse?</a></p>
</center>
```

Make it pretty. Update *layout.html*:

```
<!DOCTYPE html>
<html>
<head>
<title>check your pulse</title>
<meta charset="utf-8" />
<style type="text/css">
  body {font-family: Arial, Helvetica, sans-serif; font-size:x-large;}
</style>
{{
  middle_columns = {0:'span12',1:'span9',2:'span6'}
}}
{{block head}}{{end}}
</head>
<body>
<div class="{{=middle_columns}}>
  {{block center}}
  {{include}}
  {{end}}
</div>
</body>
</html>
```

Test it out. Is it pretty? No. It's functional:





your pulse

i really do not like css

is

neg

Another Pulse?

Make yours pretty.

check a pulse

Just another Sentiment Analysis tool.

yay

Submit

your pulse

yay

is

neutral

[Another Pulse?](#)

Better?

Next steps

What else could you do with this? Well, you could easily add a database to the application to save the inputted text as well as the results. With sentiment analysis, you want your algorithm to get smarter over time. Right now, the algorithm is static. Try entering the term "i like milk". It's negative, right?

```
{"probability": {"neg": 0.50114184747628709, "neutral": 0.34259733533730058, "pos": 0.49885815252371291}, "label": "neg"}
```

Why is that? Test out each word:

"I":

```
{"probability": {"neg": 0.54885268027242828, "neutral": 0.37816113425135217, "pos": 0.45114731972757172}, "label": "neg"}
```

"like":

```
{"probability": {"neg": 0.52484460041100567, "neutral": 0.45831376351784164, "pos": 0.47515539958899439}, "label": "neg"}
```

"milk":

```
{"probability": {"neg": 0.54015839746206784, "neutral": 0.47078672070829519, "pos": 0.45984160253793216}, "label": "neg"}
```

All negative. Doesn't seem right. The algorithm needs to be updated. Unfortunately, that's beyond the scope of this course. By saving each result in a database, you can begin analyzing the results to at least find errors and spot trends. From there, you can begin to update the algorithm. Good luck.

Sentiment Analysis Expanded

Let's take "Pulse" to the next level by adding [jQuery and AJAX](#). Don't worry if you've never worked with either jQuery or AJAX before, as web2py automates much of this. We'll also be covering both in a latter chapter. If you are interested in going through a quick primer on jQuery, check out the [Mad Libs with jQuery](#) tutorial. If you do go through the tutorial, keep in mind that you will not have to *actually* code any Javascript or jQuery in web2py for this tutorial.

Let's begin.

web2py and AJAX

web2py defines a function called `ajax()` built on top of jQuery, which essentially takes three arguments, a url, a list of ids, and a target id.

```
{{extend 'layout.html'}}
<h1>AJAX Test</h1>
<form>
  <input type="number" id="key" name ="key">
  <input type="button" value="submit"
    onclick="ajax('{{=URL('data')}}', ['key'], 'target')"
  </form>
  <br>
  <div id="target"></div>
```

Add this code to a new view in your "Pulse" app, which will be used just for testing. Call the view `test/index.html`.

This is essentially a regular form, but you can see the `ajax()` function within the button input - `onclick="ajax('{{=URL('data')}}', ['key'], 'target')`. The url will call a function within our controller (which we have yet to define), the data is grabbed from the input box, then the final results will be appended to `<div id="target"></div>`. Essentially, on the button click, we grab the value from the input box, send it to the `data()` function for *something* to happen, then it is added to the page between the `<div>` tag with the selector `id=target`.

Make sense? Let's get our controller setup. Create a new one called `test.py`:

```
def index():
    return dict()

def data():
    return (int(request.vars.key)+10)
```

So when the input data is sent to the `data()` function, we are simply adding 10 to the number and then returning it.

Update the parent template `layout.html`:

```
<!DOCTYPE html>
<head>
    <title>AJAX Test</title>
    <script src="{{=URL('static', 'js/modernizr.custom.js')}}"></script>
    <!-- include stylesheets -->
    {{{
        response.files.insert(0,URL('static','css/web2py.css'))
        response.files.insert(1,URL('static','css/bootstrap.min.css'))
        response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
        response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
    }}}
    {{include 'web2py_ajax.html'}}
    {{{
        middle_columns = {0:'span12',1:'span9',2:'span6'}
    }}}
    <noscript><link href="{{=URL('static', 'css/web2py_bootstrap_nojs.css')}}" rel="stylesheet" type="text/css" /></noscript>
    {{block head}}{{end}}
</head>
<body>
    <div class="container">
        <section id="main" class="main_row">
            <div class="{{=middle_columns}}>
                {{block center}}
                {{include}}
                {{end}}
            </div>
        </section><!--/main-->
    </div> <!-- /container -->
    <script src="{{=URL('static', 'js/bootstrap.min.js')}}"></script>
    <script src="{{=URL('static', 'js/web2py_bootstrap.js')}}"></script>
</body>
</html>
```

For more information on what is happening in this HTML, checkout the [web2py docs](#) on including jQuery and AJAX.

Test it out. Make sure to enter an integer. You should see something like this:

AJAX Test

15

25

Did you notice that when you click the button the page does not refresh? This is what makes AJAX, well, AJAX: To the end user, the process is seamless. Web apps send data from the client to the server, which is then sent back to the client without interfering with the default behavior of the browser page (no page refresh).

Let's apply this to our "Pulse" app.

Adding AJAX

Update the default controller:

```

import requests

def index():
    return dict()

def pulse():
    session.m=[]
    if request.vars.sentiment:
        text = request.vars.sentiment
        text = text.split('_')
        text = ' '.join(text)
        url = 'http://text-processing.com/api/sentiment/'
        data = {'text': text}
        r = requests.post(url, data=data)
        session.m.append(r.content)
    session.m.sort()
    return text, TABLE(*[TR(v) for v in session.m]).xml()

```

Here, we simply grab the text from the input, run the analysis, append the results to a list, and then return the original text along with the results.

Update the view, *default/index.html*:

```

{{extend 'layout.html'}}
<h1>check your pulse</h1>
<form>
    <input type="text" id="sentiment" name ="sentiment">
    <input type="button" value="submit" onclick="ajax('{{=URL('pulse')}}',['sentiment'], 'target')">
</form>
<br>
<div id="target"></div>

```

Nothing new here. Test it out. You should see something like:

check your pulse

web2py

```
{"probability": {"neg": 0.50955199890675162, "neutral": 0.59508906140405482, "pos": 0.49044800109324838}, "label": "neutral"}
```

Additional Features

Now, let's expand our app so that we can enter two text inputs for comparison.

Update the controller:

```
import requests

def index():
    return dict()

def pulse():

    session.m=[]
    url = 'http://text-processing.com/api/sentiment/'

    # first item
    text_first = request.vars.first_item
    text_first = text_first.split('_')
    text_first = ' '.join(text_first)
    data_first = {'text': text_first}
    r_first = requests.post(url, data=data_first)
    session.m.append(r_first.content)

    # second_item
    text_second = request.vars.second_item
    text_second = text_second.split('_')
    text_second = ' '.join(text_second)
    data_second = {'text': text_second}
    r_second = requests.post(url, data=data_second)
    session.m.append(r_second.content)

    session.m.sort()
    return text_first, text_second, TABLE(*[TR(v) for v in session.m]).xml()
```

This performs the exact same actions as before, only with two inputs instead of one.

Update the view:

```
 {{extend 'layout.html'}}
<h1>pulse</h1>
<p>comparisons using sentiment</p>
<br>
<form>
  <input type="text" id="first_item" name ="first_item" placeholder="enter first item...">
  <br>
  <input type="text" id="second_item" name ="second_item" placeholder="enter second item...">
  <br>
  <input type="button" value="submit" onclick="ajax('{{=URL('pulse')}}',['first_item','second_item'],'target')">
</form>

<br>
<div id="target"></div>
```

Notice how we're passing two items to the pulse URL, `['first_item', 'second_item']`. Test.

pulse

comparisons using sentiment

ruby

python

submit

rubypython

```
{"probability": {"neg": 0.42554563711374715, "neutral": 0.7963320491099668, "pos": 0.57445436288625285}, "label": "neutral"}
{"probability": {"neg": 0.50471472774640502, "neutral": 0.59508906140405482, "pos": 0.49528527225359503}, "label": "neutral"}
```

Now, let's make this look a little nicer.

Only display the item that has the higher (more positive) sentiment. To do this, update the controller. Go to the assets file in the [book-2 exercises repo](#).

Update, the controller, `default.py`, with the code from the repo.

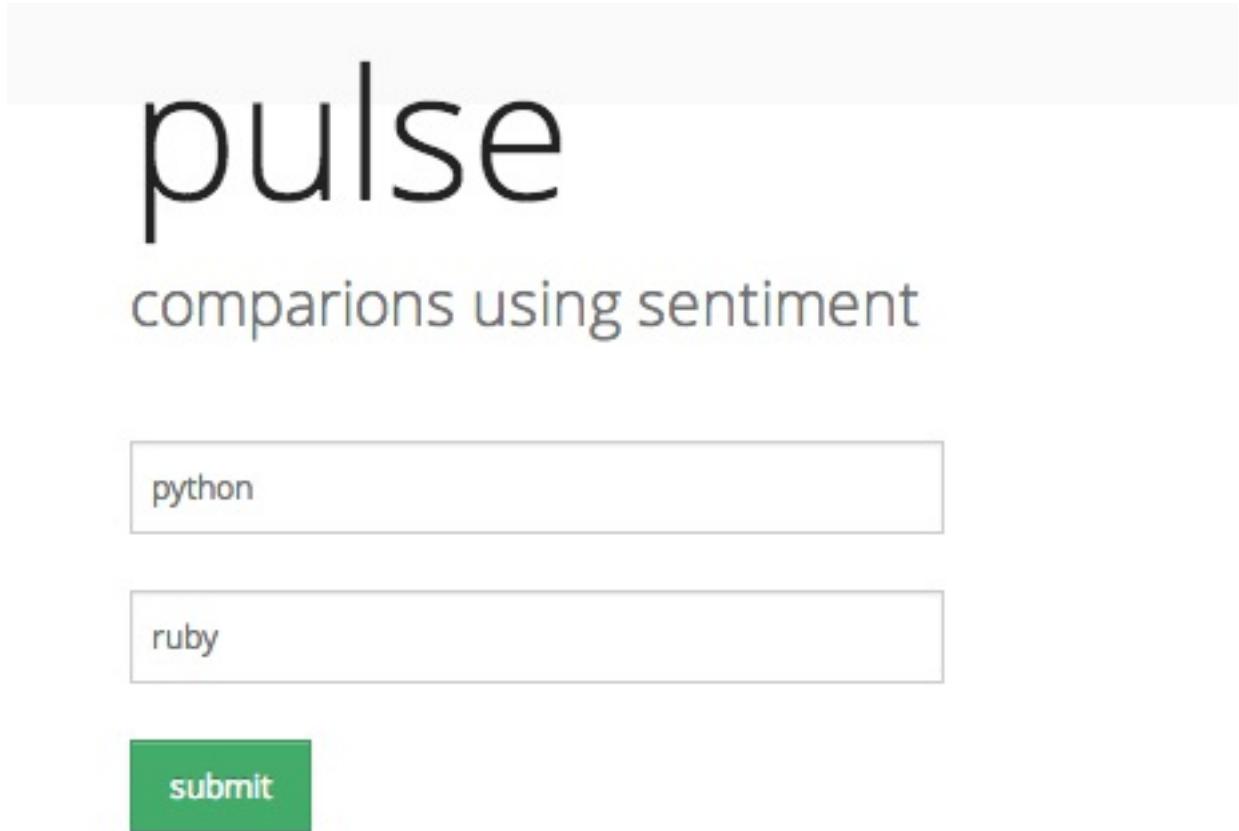
Although there is quite a bit more code here, the logic is simple. Walk through the program, slowly, stating what each line is doing. Do this out loud. Keep going through it until it makes sense. Also, make sure to test it to ensure it's returning the right values. You can compare the results with the results found [here](#).

Next, let's clean up the output. Go to the assets file in the [book-2 exercises repo](#).

First, update the layout template with a Bootstrap stylesheet as well as some custom styles.

Next, update the child template, *default.index.html*.

We simply added some bootstrap classes to make it look much nicer.



pulse

comparions using sentiment

python

ruby

submit

The winner is python!

Looking good! Nice Job.

Movie Suggester

Let's take this to the next level and create a movie suggester app. Essentially, we'll pull data from the [My API Films](#) API that we used earlier to grab movies that are playing then display the sentiment of each.

Setup

Create a new app called "movie_suggest". Replace the code in the controller, *default.py* with:

```
import requests
import json

def index():
    return dict()

def grab_movies():
    session.m = []
    TOKEN = 'GET_YOUR_OWN_KEY'
    requests.get('http://api.myapifilms.com/imdb/inTheaters?token=' +
        '{0}&format=json&language=en-us'.format(TOKEN))
    binary = url.content
    output = json.loads(binary)
    movies = output['data']['inTheaters']
    for movie in movies:
        all_movies = movie['movies']
        for meta in all_movies:
            if(meta['title']):
                session.m.append(meta["title"])
    session.m.sort()
    return TABLE(*[TR(v) for v in session.m]).xml()
```

NOTE: Make sure you add your API key into the value for the variable `TOKEN`.

In this script, we're using the My API Films to grab the movies currently playing. Test this out in your shell to see exactly what's happening:

```
>>> import requests
>>> import json
>>> TOKEN = 'abha3gx3p42czdrswkejmyxm'
>>> url = requests.get('http://api.myapifilms.com/imdb/inTheaters?token=' +
    '{0}&format=json&language=en-us'.format(TOKEN))
>>> binary = url.content
>>> output = json.loads(binary)
>>> movies = output['data']['inTheaters']
>>> for movie in movies:
...     all_movies = movie['movies']
...     for meta in all_movies:
...         if meta['title']:
...             print(meta['title'])
Oppression
Billy Lynn's Long Halftime Walk
Almost Christmas
Arrival
Elle
The Love Witch
Doctor Strange
Trolls
Boo! A Madea Halloween
Inferno
Tu ne tueras point
The Girl on the Train
The Accountant
Miss Peregrine's Home for Peculiar Children
Ouija: les origines
Jack Reacher: Never Go Back
```

Update the view, *default/index.html*:

```
{{extend 'layout.html'}}
<h1>suggest-a-movie</h1>
<p>use sentiment to find that perfect movie</p>
<br>
<form>
    <input type="button" value="Get Movies" onclick="ajax('{{=URL('grab_movies')}}',
[], 'target')">
</form>
<br>
<div id="target"></div>
```

Update the parent view, *layout.html*:

```
<!DOCTYPE html>
<head>
<title>suggest-a-movie</title>
<script src="{{=URL('static','js/modernizr.custom.js')}}"></script>
<!-- include stylesheets -->
{{{
response.files.insert(0,URL('static','css/web2py.css'))
response.files.insert(1,URL('static','css/bootstrap.min.css'))
response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
}}}
{{include 'web2py_ajax.html'}}
{{{
middle_columns = {0:'span12',1:'span9',2:'span6'}
}}}
<link href="http://maxcdn.bootstrapcdn.com/
  bootstrap/3.2.0/yeti/bootstrap.min.css" rel="stylesheet">
<noscript><link href="{{=URL('static', 'css/web2py_bootstrap_nojs.css')}}" rel="stylesheet" type="text/css" /></noscript>
{{block head}}{{end}}
</head>
<body>
<div class="container">
<div class="jumbotron">
<div class="{{=middle_columns}}>
  {{block center}}
  {{include}}
  {{end}}
</div>
</div>
</div> <!-- /container -->
<script src="{{=URL('static','js/bootstrap.min.js')}}"></script>
<script src="{{=URL('static','js/web2py_bootstrap.js')}}"></script>
</body>
</html>
```

Test it out. You should see something similar to this (depending on which movies are in the theater, of course):

suggest-a-movie

use sentiment to find that perfect movie

Get Movies

3 Days To Kill
About Last Night
American Hustle
Anchorman 2: The Legend Continues
Endless Love
Lone Survivor
Non-Stop
Pompeii
Ride Along
RoboCop
Son Of God
The Lego Movie
The Monuments Men
The Nut Job
The Wind Rises
Winter's Tale

Add Sentiment

Let's now determine the sentiment of each movie. Update the controller:

```

import requests
import json

def index():
    return dict()

def grab_movies():
    session.m = []
    TOKEN = 'GET_YOUR_OWN_KEY'
    requests.get('http://api.myapifilms.com/imdb/inTheaters?token=' +
                 '{0}&format=json&language=en-us'.format(TOKEN))
    binary = url.content
    output = json.loads(binary)
    movies = output['data']['inTheaters']
    for movie in movies:
        all_movies = movie['movies']
        for meta in all_movies:
            if(meta['title']):
                session.m.append(pulse(meta["title"]))
    session.m.sort()
    return TABLE(*[TR(v) for v in session.m]).xml()

def pulse(movie):
    text = movie.replace('_', ' ')
    url = 'http://text-processing.com/api/sentiment/'
    data = {'text': text}
    r = requests.post(url, data=data)
    binary = r.content
    output = json.loads(binary)
    label = output["label"]
    pos = output["probability"]["pos"]
    neg = output["probability"]["neg"]
    neutral = output["probability"]["neutral"]
    return text, label, pos, neg, neutral

```

Here we are simply taking each name and passing them as an argument into the `pulse()` function where we are calculating the sentiment.

suggest-a-movie

use sentiment to find that perfect movie

Get Movies

3 Days To Kill	neutral	0.3021240014340.6978759985660.96141481597
About Last Night	neutral	0.4894781079380.5105218920620.777012481552
American Hustle	neutral	0.6184005631970.3815994368030.663263611318
Anchorman 2: The Legend Continues	neutral	0.47670093557 0.52329906443 0.800510393573
Endless Love	pos	0.5229853853920.4770146146080.462337287624
Lone Survivor	neutral	0.5241479270120.4758520729880.757142151876
Non-Stop	neutral	0.4316059211660.5683940788340.7897784246
Pompeii	neutral	0.4904480010930.5095519989070.595089061404
Ride Along	neutral	0.6104252652 0.3895747348 0.538105042179
RoboCop	neutral	0.5026478605770.4973521394230.595089061404
Son Of God	neutral	0.5683231581460.4316768418540.881602969824
The Lego Movie	neg	0.4881549487380.5118450512620.40276071531
The Monuments Men	neutral	0.5057370719790.4942629280210.681055003825
The Nut Job	neutral	0.5820900141920.4179099858080.774520853746
The Wind Rises	neutral	0.43051395448 0.56948604552 0.552275748981
Winter's Tale	pos	0.5909380730380.4090619269620.417161505192

Update Styles

Update the child view with some bootstrap styles:

```
 {{extend 'layout.html'}}
<h1>suggest-a-movie</h1>
<p>use sentiment to find that perfect movie</p>
<br>
<form>
  <input type="button" class="btn btn-primary" value="Get Movies" onclick="ajax('{=URL('grab_movies')}',[], 'target')">
</form>
<br>
<table class="table table-hover">
  <thead>
    <tr>
      <th>Movie Title</th>
      <th>Label</th>
      <th>Positive</th>
      <th>Negative</th>
      <th>Neutral</th>
    </tr>
  </thead>
  <tbody id="target"></tbody>
</table>
```

suggest-a-movie

use sentiment to find that perfect movie

[Get Movies](#)

Movie Title	Label	Positive	Negative	Neutral
3 Days To Kill	neutral	0.302124001434	0.697875998566	0.96141481597
About Last Night	neutral	0.489478107938	0.510521892062	0.777012481552
American Hustle	neutral	0.618400563197	0.381599436803	0.663263611318
Anchorman 2: The Legend Continues	neutral	0.47670093557	0.52329906443	0.800510393573
Endless Love	pos	0.522985385392	0.477014614608	0.462337287624
Lone Survivor	neutral	0.524147927012	0.475852072988	0.757142151876
Non-Stop	neutral	0.431605921166	0.568394078834	0.7897784246
Pompeii	neutral	0.490448001093	0.509551998907	0.595089061404
Ride Along	neutral	0.6104252652	0.3895747348	0.538105042179
RoboCop	neutral	0.502647860577	0.497352139423	0.595089061404
Son Of God	neutral	0.568323158146	0.431676841854	0.881602969824

Think about what else you could do? Perhaps you could highlight movies that are positive. Check the [web2py documentation](#) for help.

Deploy to Heroku

Open `setup-web2py-heroku.sh` and comment out the line:

Then in the `.gitignore`, which is in the root directory, comment out this line:

```
applications/*/private/*
```

And then comment out this line in `scripts/setup-web2py-heroku.sh`:

```
pip freeze > requirements.txt
```

So now the file will look like this:

```
read -p "Choose your admin password?" passwd
sudo pip install virtualenv
virtualenv venv --distribute
source venv/bin/activate
sudo pip install psycopg2
# pip freeze > requirements.txt
echo "web: python web2py.py -a '$passwd' -i 0.0.0.0 -p \$PORT" > Procfile
git init
git add .
git add Procfile
git commit -a -m "first commit"
heroku create
git push heroku master
heroku addons:add heroku-postgresql:dev
heroku scale web=1
heroku open
```

As you can see web2py is using *virtualenv* to manage it's environment. Because we are using pyenv we have to make sure that this deploy process installs the proper requirements for our applications. So, now we need to run:

```
$ pyenv exec pip freeze > requirements.txt
```

This will write all of our needed packages to the requirements file allowing the web2py deploy script to install them for us.

Finally to deploy, we will run the script provided by web2py, it should work like a charm with those fews modifications. Run this:

```
$ sudo sh scripts/setup-web2py-heroku.sh
```

This will perform all the steps necessary to create, install and fire up your app, just sit back and watch!

Check out the live app at https://still-sierra-82618.herokuapp.com/movie_suggest/default/index.

Cheers!

web2py: py2manager

In the last chapters we built several small applications to illustrate the power of web2py. Those applications were meant more for learning. In this chapter we will develop a much larger application - a task manager, similar to FlaskTaskr, called **py2manager**.

This application will be developed from the ground up to not only show you all that web2py has to offer - but to also dig deeper into modern web development and the Model View Controller pattern.

This application will do the following:

1. Users must sign in (and register, if necessary) before hitting the landing page, *index.html*.
2. Once signed in, users can add new companies, notes, and other information associated with a particular project and view other employees' profiles.

Regarding the actual data model:

1. Each `company` consists of a company name, email, phone number, and URL.
2. Each `project` consists of a name, employee name (person who logged the project), description, start date, due date, and completed field that indicates whether the project has been completed.
3. Finally, the `notes` reference a project and include a text field for the actual note, created date, and a created by field.

Up to this point, you have developed a number of different applications using the Model View Controller (MVC) architecture pattern:

1. **Model:** data warehouse (database), source of "truth"
2. **View:** data output
3. **Controller:** link between the user and the application

Again, a user sends a request to a web server. The server, in turn, passes that request to the controller. The controller then performs an action, such as querying (GET) or modifying (POST, PUT, DELETE) the database. Once the data is found or modified, the controller then sends a response back to the views, which are rendered into HTML for the end the user to see.

Most modern web frameworks utilize the MVC-style architecture, offering similar components. Each framework implements the various components slightly different, due to the choices made by the developers of the framework. Learning about such differences is vital for sound development.

We'll look at MVC in terms of web2py as we develop our application.

Setup

Lets try yet another way of installing the web2py app on our local machine:

1. Create a new directory called "py2manager" to house your app.
2. Download the source code from the web2py [repository](#) and place it into the "py2manager" directory. Unzip the file, placing all files and folders into the "py2manager" directory.
3. Change directories into `web2manager`.
4. Using `pyenv` , set the local version of Python: `pyenv local 2.7.6`
5. Create a new app from your terminal: `pyenv exec python web2py.py -S py2manager` .
After this project is created, we are left in the web2py Shell. Go ahead and exit it.
6. Within the terminal, navigate to the "applications" directory, then into the "py2manager" directory. This directory holds all of your application's files.

Text Editor

Instead of using the internal web2py IDE for development, like in the previous examples, let's jump back to traditional development and use a text editor, like Sublime Text. Load the entire project into your editor.

You should already have Sublime Text installed from previous work we have done. Checkout the *Getting Started* chapter of this book for instructions on installing and configuring Sublime Text.

In your console, from `py2manager/`, run:

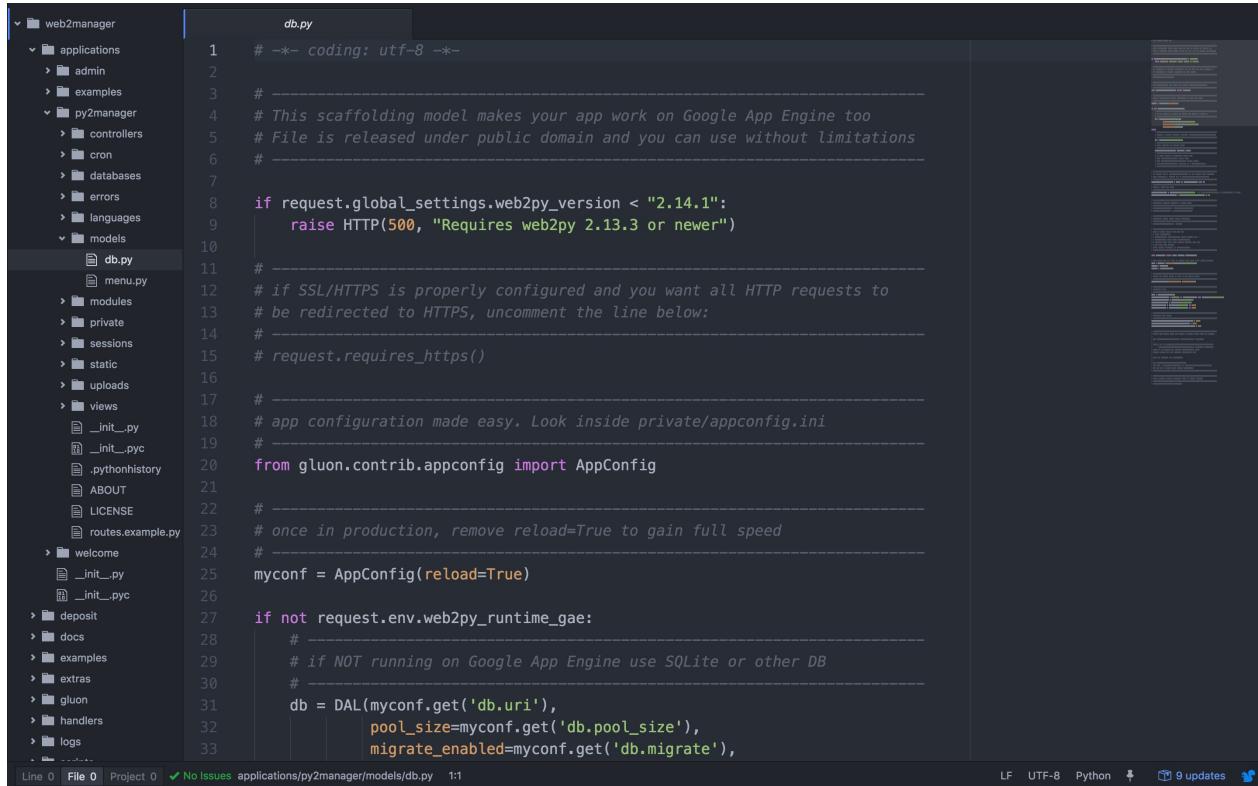
```
$ subl .
```

If that command is not found, run this snippet to create enable this short cut:

```
$ sudo ln -s /Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl /usr/bin/subl
```

Try again, this should load all of the files into Sublime text and open a sublime text window. For more information on setting up Sublime Text for Python web development checkout [this blog post](#).

You should now have the entire application structure (files and folders) in Sublime. Take a look around. Open the "Models", "Views", and "Controllers" folders. Simply double-click to open a particular file to load it in the editor window. Files appear as tabs on the top of the editor window, allowing you to move between them quickly.



The screenshot shows the Sublime Text interface with the following details:

- File Explorer:** On the left, the project structure is displayed under the root folder "web2manager". The "models" folder is currently selected, showing its contents: db.py, menu.py, modules, private, sessions, static, uploads, and views. Other files like __init__.py, __init__.pyc, .pythonhistory, ABOUT, LICENSE, and routes.example.py are also listed.
- Editor:** The main window displays the content of the "db.py" file. The code is as follows:

```
1  # -*- coding: utf-8 -*-
2
3  #
4  # -----
5  # This scaffolding model makes your app work on Google App Engine too
6  # File is released under public domain and you can use without limitations
7  #
8  if request.global_settings.web2py_version < "2.14.1":
9      raise HTTP(500, "Requires web2py 2.13.3 or newer")
10
11 #
12 # if SSL/HTTPS is properly configured and you want all HTTP requests to
13 # be redirected to HTTPS, uncomment the line below:
14 #
15 # request.requires_https()
16
17 #
18 # app configuration made easy. Look inside private/appconfig.ini
19 #
20 from gluon.contrib.appconfig import AppConfig
21
22 #
23 # once in production, remove reload=True to gain full speed
24 #
25 myconf = AppConfig(reload=True)
26
27 if not request.env.web2py_runtime_gae:
28     #
29     # if NOT running on Google App Engine use SQLite or other DB
30     #
31     db = DAL(myconf.get('db.uri'),
32             pool_size=myconf.get('db.pool_size'),
33             migrate_enabled=myconf.get('db.migrate'))
```

- Bottom Status Bar:** Shows "Line 0 | File 0 | Project 0 | No issues | applications/py2manager/models/db.py | 1:1".
- Bottom Right:** Shows "LF | UTF-8 | Python" and a "9 updates" badge.

Let's start developing!

Database

As you saw in the previous chapter, web2py uses an API called a Database Abstraction Layer (DAL) to map Python objects to database objects. Like an ORM, a DAL hides the complexity of the underlying SQL code. The major [difference](#) between an ORM and a DAL, is that a DAL operates at a lower level. In other words, its syntax is somewhat closer to SQL. If you have experience with SQL, you may find a DAL easier to work with than an ORM. If not, learning the syntax is no more difficult than an ORM.

ORM:

```
class User(db.Model):
    __tablename__ = 'users'
    name = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
    password = db.Column(db.String, nullable=False)
```

DAL:

```
db.define_table('users',
    Field('name', 'string', unique=True, notnull=True),
    Field('email', 'string', unique=True, notnull=True),
    Field('password', 'string', 'password', readable=False, label='Password'))
```

The above examples create the exact same "users" table. ORMs generally use classes to declare tables, while the web2py DAL flavor uses functions. Both are portable among many different relational database engines. Meaning they are database agnostic, so you can switch your database engine without having to re-write the code within your Model. web2py is integrated with a number of popular databases, including SQLite, PostgreSQL, MySQL, SQL Server, FireBird, Oracle, MongoDB, among others.

Shell

If you prefer the command line, you can work directly with your database from the web2py Shell. The following is a quick, unrelated example...

In your terminal navigate to the project root directory, "/web2py/py2manager", and then run the following command:

```
$ pyenv exec python web2py.py --shell=py2manager
```

Run the following DAL [commands](#):

```
>>> db = DAL('sqlite://storage.sqlite', pool_size=1, check_reserved=['all'])
>>> db.define_table('special_users', Field('name'), Field('email'))
<Table special_users (id, name, email)>
>>> db.special_users.insert(id=1, name="Alex", email="hey@alex.com")
1L
>>> db.special_users.bulk_insert([{'name':'Alan', 'email':'a@a.com'}, {'name':'John', 'email':'j@j.com'}, {'name':'Tim', 'email':'t@t.com'}])
[2L, 3L, 4L]
>>> db.commit()
>>> for row in db().select(db.special_users.ALL):
...     print row.name
...
Alex
Alan
John
Tim
>>> for row in db().select(db.special_users.ALL):
...     print row
...
<Row {'name': 'Alex', 'email': 'hey@alex.com', 'id': 1}>
<Row {'name': 'Alan', 'email': 'a@a.com', 'id': 2}>
<Row {'name': 'John', 'email': 'j@j.com', 'id': 3}>
<Row {'name': 'Tim', 'email': 't@t.com', 'id': 4}>
>>> db.special_users.drop()
>>> exit()
```

Here, we created a new table called "special_users" with the fields "name" and "email". We then inserted a single row of data, then multiple rows. Finally, we printed the data to the screen using for loops before dropping (deleting) the table and exiting the Shell.

web2py Admin

Now, as we mentioned before, web2py has a default for everything. These are the default values for each table field:

```
Field(name, 'string', length=None, default=None,
    required=False, requires='<default>',
    ondelete='CASCADE', notnull=False, unique=False,
    uploadfield=True, widget=None, label=None, comment=None,
    writable=True, readable=True, update=None, authorize=None,
    autodelete=False, represent=None, compute=None,
    uploadfolder=os.path.join(request.folder, 'uploads'),
    uploadseparate=None, uploadfs=None)
```

So, our `company_name` field would by default be a string value, it is not required (meaning it is not mandatory to enter a company name), and, finally, it does not have to be a unique value. Keep these defaults in mind when you are creating your database tables, as you will often need to override them.

Let's create the model for our application. Navigate into the "applications" directory, and then to the "py2manager" directory. Create a new file to define your database schema called `db_tasks.py` within the "Models" directory.

Add the following code:

```
db.define_table('company',
    Field('company_name', notnull=True, unique=True),
    Field('email'),
    Field('phone', notnull=True),
    Field('url'),
    format = '%(company_name)s')

db.company.email.requires=IS_EMAIL()
db.company.url.requires=IS_EMPTY_OR(IS_URL())

db.define_table('project',
    Field('name', notnull=True),
    Field('employee_name', db.auth_user, default=auth.user_id),
    Field('company_name', 'reference company', notnull=True),
    Field('description', 'text', notnull=True),
    Field('start_date', 'date', notnull=True),
    Field('due_date', 'date', notnull=True),
    Field('completed', 'boolean', notnull=True),
    format = '%(company_name)s')

db.project.employee_name.readable = db.project.employee_name.writable = False
```

We defined two tables tables: `company` and `project`. You can see the foreign key in the `project` table, `reference company`. The `auth_user` table is an auto-generated table. Also, the `employee_name` field in the `project` table references the logged in user. So

when a user posts a new project, the user information will automatically be added to the database. Save the file.

Navigate back to your project root. Fire up web2py in your terminal:

```
$ pyenv exec python web2py.py -a 'PUT_YOUR_PASSWORD_HERE' -i 127.0.0.1 -p 8000
```

NOTE: Make sure to replace 'PUT_YOUR_PASSWORD_HERE' with an actual password - e.g., `python web2py.py -a admin -i 127.0.0.1 -p 8000`

Navigate to <http://localhost:8000/> in your browser. Make your way to the **Edit** page and click the "database administration" button to execute the DAL commands. Take a look at the `sql.log` file within the "databases" directory in Sublime to verify exactly which tables and fields were created.

Have look in the `assets` folder of the book 2 exercises [repo](#) and compare the `sql.log` to your file and make sure they match up.

SEE ALSO: You can also read the documentation on all the auto-generated tables in the web2py official [documentation](#).

Notice the `format` attribute. All references are linked to the Primary Key of the associated table, which is the auto-generated ID. By using the `format` attribute, references will not show up by the `id` - but by the preferred field. You'll see *exactly* what that means in a second.

Open <http://localhost:8000/py2manager/default/user/register>, then register yourself as a new user.

Next, let's setup a new company and an associated project by navigating to <http://localhost:8000/py2manager/appadmin/index>. Click the relevant tables and add in data for the company and project. Make sure *not* to mark the project as complete.

NOTE: web2py addresses a number of potential security flaws automatically. One of them is session management: *web2py provides a built-in mechanism for administrator authentication, and it manages sessions independently for each application. The administrative interface also forces the use of secure session cookies when the client is not "localhost".*

One less thing you have to worry about. For more information, please check out the web2py [documentation](#).

Homework

- Download the [web2py cheatsheet](#). Read it.

URL Routing

Controllers describe the application/business logic and workflow in order to link the user with the application through the request/response cycle. More precisely, the controller controls the requests made by the users, obtains and organizes the desired information, and then responds back to the user via views and templates.

For example, navigate to the login [page](#) within your app and log in with the user that you created. When you clicked the "Login" button after you entered your credentials, a POST request was sent to the controller. The controller then took that information, and compared it with the users in the database via the model. Once your user credentials were found, this information was sent back to the controller. Then the controller redirected you to the appropriate view.

Web frameworks simplify this process significantly.

URL Routing with web2py

web2py provides a simple [means](#) of matching URLs with views. In other words, when the controller provides you with the appropriate view, there is a URL associated with that view, which can be customized.

Let's look at an example:

```
def index():
    return dict(message="Hello!")
```

This is just a simple function used to output the string "Hello!" to the screen. The function name is `index()`. You can't tell from the above info, but the application name is "hello" and the controller used for this function is *default.py*.

In this case the generated URL will be:

<http://www.yoursite.com/hello/default/index.html>

This is also the default URL routing method:

http://www.yoursite.com/application_name/controller_name/function_name.html

You can customize the URL routing methods by adding a *routes.py* file to "web2py/py2manager" (the outer "py2manager" directory). For example, to remove the "controller_name" from the URL, add the following code to the newly created file:

```
routers = dict(
    BASE = dict(
        default_application='py2manager',
    )
)
```

Make those changes.

Test this out. Restart the server. Navigate to the login page again:

<http://localhost:8000/py2manager/default/user/login>. Well, since we made those

changes, we can now access the same page from this url

<http://localhost:8000/user/login>.

SEE ALSO: For more information on URL routing, please see the official web2py documentation.

Let's configure the logic and URL routing in the py2manager app. Add the following code to *default.py*:

```
@auth.requires_login()
def index():
    project_form = SQLFORM(db.project).process()
    projects = db(db.project).select()
    users = db(db.auth_user).select()
    companies = db(db.company).select()
    return locals()
```

Here we displayed the data found in the `project`, `auth_user`, and `company` tables, as well as added a form for adding projects. With that, most of the functionality is in place. We just need to update the views, organize the *index.html* page, and update the layout and styles.

Initial Views

Views (or templates) describe how the subsequent response, from a request, should be translated to the user using mostly a combination of a templating engine, HTML, Javascript, and CSS.

Some major components of the templates include...

Template Engine

Template engines are used for embedding Python code directly into standard HTML. web2py uses a slightly modified Python syntax to make the code more readable. You can also define control statements such as `for` and `while` loops as well as `if` statements.

Let's look at a quick example

Add a basic function to the controller:

```
def tester():
    return locals()
```

Next, create a new view, `default/tester.html`:

```
<html>
  <body>
    {{numbers = [1, 2, 3]}}
    <ul>
      {{for n in numbers:}}<li>{{=n}}</li>{{/pass}}
    </ul>
  </body>
</html>
```

Test it out <http://localhost:8000/tester>.

Template Composition

Like most templating languages, the web2py flavor can extend and include a set of sub templates. For example, you could have the base or child template, `index.html`, that extends from a parent template, `default.html`. Meanwhile, `default.html` could include two

sub templates, *header.html* and *footer.html*:

For example, add the parent template:

```
 {{extend 'layout.html'}}  
<html>  
  <body>  
    {{numbers = [1, 2, 3]}}  
    <ul>  
      {{for n in numbers:}}<li>{{=n}}</li>{{/pass}}  
    </ul>  
  </body>  
</html>
```

Test it out again at <http://localhost:8000/tester>.

JavaScript Libraries

As you have seen, web2py includes a number of JavaScript libraries, many of which are pre-configured. Refer to the web2py [documentation](#) for more information on JavaScript, jQuery, and other components of the views.

With that, let's build the templates for py2manager...

Templates

default/index.html:

```
{{extend 'layout.html'}}
<h2>Welcome to py2manager</h2>
<br/>
{{=(project_form)}}
<br/>
<h3> All Open Projects </h3>
<ul>{{for project in projects:}}
<li>
{{=(project.name)}}
</li>
{{pass}}
</ul>
```

This file has a form at the top to add new projects to the database. It also lists out all open projects using a `for` loop. You can view the results here:

<http://localhost:8000/index>. Notice how this template extends from *layout.html*.

default/user.html:

Open up this file. This template was created automatically to make the development process easier and quicker. If you go back to the *default.py* file, you can see a description of the main functionalities of the user function:

```
"""
exposes:
http://..../[app]/default/user/login
http://..../[app]/default/user/logout
http://..../[app]/default/user/register
http://..../[app]/default/user/profile
http://..../[app]/default/user/retrieve_password
http://..../[app]/default/user/change_password
use @auth.requires_login()
    @auth.requires_membership('group name')
    @auth.requires_permission('read','table name',record_id)
to decorate functions that need access control
"""
```

Read more about authentication [here](#).

Let's edit the main layout to replace the generic template. Start with *models/menu.py*.

Update the following code:

```
response.logo = A(B('py',SPAN(2), 'manager'),_class="brand")
response.title = "py2manager"
response.subtitle = T('just another project manager')
```

Then update the application menu:

```
response.menu = [(T('Home'), False, URL('default', 'index'), []),
(T('Add Project'), False, URL('default', 'add'), []),
(T('Add Company'), False, URL('default', 'company'), []),
(T('Employees'), False, URL('default', 'employee'), [])]

DEVELOPMENT_MENU = False
```

Take a look at your changes:

```
#####
## Customize your APP title, subtitle and menus here
#####

response.logo = A(B('py',SPAN(2), 'manager'),_class="brand")
response.title = "py2manager"
response.subtitle = T('just another project manager')

#####
## this is the main application menu add/remove items as required
#####

response.menu = [(T('Home'), False, URL('default', 'index'), []),
(T('Add Project'), False, URL('default', 'add'), []),
(T('Add Company'), False, URL('default', 'company'), []),
(T('Employees'), False, URL('default', 'employee'), [])]

DEVELOPMENT_MENU = False
```

Now that we've gone over the Model View Controller architecture, let's shift to focus on the main functionality of the application.

Profile Page

Remember the auto-generated `auth_user` table? Take a look at the `sql.log` for a quick reminder. In short, the `auth_user` table is part of a larger set of auto-generated tables aptly called the [Auth](#) tables.

It's easy to add fields to any of the Auth tables. Open up `db.py` and place the following code after `auth = Auth(db)` and before `auth.define_tables()` (line 90, at time of writing - 11/26/2016):

```
auth.settings.extra_fields['auth_user'] = [
    Field('address'),
    Field('city'),
    Field('zip'),
    Field('image', 'upload')]
```

Save the file. Navigate to <http://localhost:8000/index> and log in if necessary. Once logged in, you can see your name in the upper right-hand corner. Click the drop down arrow, then select "Profile". You should see the new fields. Go ahead and update them and upload an image. Then click "Save Profile". Nice, right?

Profile

First name	<input type="text" value="pete"/> 
Last name	<input type="text" value="pete"/>
E-mail	<input type="text" value="pete@pete.com"/>
Address	<input type="text" value="111 Colfax St."/>
City	<input type="text" value="Denver"/>
Zip	<input type="text" value="80206"/>
Image	<p>Choose File profile-pic.jpg [file <input type="checkbox"/> delete]</p> 
<input type="button" value="APPLY CHANGES"/>	

Add Projects

To clean up the homepage, let's move the form to add new projects to a separate page. Open your `controllers/default.py` file, and add a new function:

```
@auth.requires_login()
def add():
    project_form = SQLFORM(db.project).process()
    return dict(project_form=project_form)
```

Then update the `index()` function like so:

```
@auth.requires_login()
def index():
    projects = db(db.project).select()
    users = db(db.auth_user).select()
    companies = db(db.company).select()
    return locals()
```

Add a new template in the default directory (`views/default/`) called `add.html`:

```
{{extend 'layout.html'}}
<h2>Add a new project:</h2>
<br/>
{{=project_form.custom.begin}}
<strong>Project name</strong><br/>{{=project_form.custom.widget.name}}<br/>
<strong>Company name</strong><br/>{{=project_form.custom.widget.company_name}}<br/>

<strong>Description</strong><br/>
{{=project_form.custom.widget.description}}<br/>
<strong>Start Date</strong><br/>{{=project_form.custom.widget.start_date}}<br/>
<strong>Due Date</strong><br/>{{=project_form.custom.widget.due_date}}<br/>
{{=project_form.custom.submit}}
{{=project_form.custom.end}}
```

In the controller, we used web2py's `SQLFORM` to generate a form automatically from the database. We then customized the look of the form using the following syntax:

```
form.custom.widget[fieldname] .
```

Remove the form from `index.html`:

```
 {{extend 'layout.html'}}
<h2>Welcome to py2manager</h2>
<br>
<h3> All Open Projects </h3>
<ul>{{for project in projects:}}
<li>
  {{=(project.name)}}
</li>
  {{pass}}
</ul>
```

Add Companies

We need to add a form for adding new companies, which follows nearly identical pattern as adding a form for projects. Try working on it on your own before looking at the code.

default.py:

```
@auth.requires_login()
def company():
    company_form = SQLFORM(db.company).process()
    return dict(company_form=company_form)
```

Add a new template in the default directory called *company.html*:

```
{{extend 'layout.html'}}
<h2>Add a new company:</h2>
<br/>
{{=company_form.custom.begin}}
<strong>Company Name</strong><br/>{{=company_form.custom.widget.company_name}}<br/>
<strong>Email</strong><br/>{{=company_form.custom.widget.email}}<br/>
<strong>Phone</strong><br/>{{=company_form.custom.widget.phone}}<br/>
<strong>URL</strong><br/>{{=company_form.custom.widget.url}}<br/>
{{=company_form.custom.submit}}
{{=company_form.custom.end}}
```

Homepage

Now, let's finish organizing the homepage to display all projects. We'll be using the `SQLFORM.grid` to display all projects. Essentially, the `SQLFORM.grid` is a high-level table that creates complex CRUD controls. It provides pagination, the ability to browse, search, sort, create, update and delete records from a single table.

Update the `index()` function in `default.py`:

```
@auth.requires_login()
def index():
    response.flash = T('Welcome!')
    grid = SQLFORM.grid(db.project)
    return locals()
```

`return locals()` is used to return a dictionary to the view, containing all the variables. It's equivalent to `return dict(grid=grid)`, in the above example. We also added a flash greeting.

Update the `index.html` view:

```
{{extend 'layout.html'}}
<h2>All projects:</h2>
<br/>
{{=grid}}
```

Navigate to <http://127.0.0.1:8000/py2manager/default> to view the new layout. Play around with it. Add some more projects. Download them in CSV. Notice how you can sort specific fields in ascending or descending order by clicking on the header links. This is the generic grid. Let's customize it to fit our needs.

Append the following code to the bottom of `db_tasks.py`:

```
db.project.start_date.requires = IS_DATE(format=T('%m-%d-%Y'),
                                         error_message='Must be MM-DD-YYYY!')
db.project.due_date.requires = IS_DATE(format=T('%m-%d-%Y'),
                                         error_message='Must be MM-DD-YYYY!')
```

This changes the date format from YYYY-MM-DD to MM-DD-YYYY. What happens if you use a lowercase `y` instead? Try it and see.

Test this out by adding a new project: <http://127.0.0.1:8000/py2manager/default/add>. Use the built-in AJAX calendar. *Oops!* That's still inputting dates the old way. Let's fix that.

Within the "views" folder, open `web2py_ajax.html` and change:

```
var w2p_ajax_date_format = "{{=T('%Y-%m-%d')}}";
var w2p_ajax_datetime_format = "{{=T('%Y-%m-%d %H:%M:%S')}}";
```

To:

```
var w2p_ajax_date_format = "{{=T('%m-%d-%Y')}}";
var w2p_ajax_datetime_format = "{{=T('%m-%d-%Y %H:%M:%S')}}";
```

Now let's update the grid in the `index()` function within the controller:

```
grid = SQLFORM.grid(db.project, create=False,
    fields=[db.project.name, db.project.employee_name,
    db.project.company_name, db.project.start_date,
    db.project.due_date, db.project.completed],
    deletable=False, maxtextlength=50)
```

What does this do? Take a look at the documentation [here](#). It's all self-explanatory. Compare the before and after output for additional help.

More Grids

First, let's add a grid to the company view.

default.py:

```
@auth.requires_login()
def company():
    company_form = SQLFORM(db.company).process()
    grid = SQLFORM.grid(db.company, create=False, deletable=False, editable=False,
    maxtextlength=50, orderby=db.company.company_name)
    return locals()
```

company.html:

```
{{extend 'layout.html'}}
<h2>Add a new company:</h2>
<br/>
{{=company_form.custom.begin}}
<strong>Company Name</strong><br/>{{=company_form.custom.widget.company_name}}<br/>

<strong>Email</strong><br/>{{=company_form.custom.widget.email}}<br/>
<strong>Phone</strong><br/>{{=company_form.custom.widget.phone}}<br/>
{{=company_form.custom.submit}}
{{=company_form.custom.end}}
<br/>
<br/>
<h2>All companies:</h2>
<br/>
{{=grid}}
```

Next, let's create the employee view:

default.py:

```
@auth.requires_login()
def employee():
    employee_form = SQLFORM(db.auth_user).process()
    grid = SQLFORM.grid(db.auth_user, create=False, fields=[db.auth_user.first_name,
    db.auth_user.last_name, db.auth_user.email], deletable=False, editable=False,
    maxtextlength=50)
    return locals()
```

employee.html:

```
 {{extend 'layout.html'}}  
<h2>All employees:</h2>  
<br/>  
{ { =grid}}
```

Test both of the new views in the browser.

Notes

Next, let's add the ability to add notes to each project. Add a new table to the database in `db_task.py`:

```
db.define_table('note',
    Field('post_id', 'reference project', writable=False),
    Field('post', 'text', notnull=True),
    Field('created_on', 'datetime', default=request.now, writable=False),
    Field('created_by', db.auth_user, default=auth.user_id))

db.note.post_id.readable = db.note.post_id.writable = False
db.note.created_on.readable = db.note.created_on.writable = False
db.note.created_on.requires = IS_DATE(format=T('%m-%d-%Y'),
    error_message='Must be MM-DD-YYYY!')
db.note.created_by.readable = db.note.created_by.writable = False
```

Update the `index()` function and add a `note()` function in the controller:

```
@auth.requires_login()
def index():
    response.flash = T('Welcome!')
    notes = [lambda project: A('Notes', _href=URL("default", "note", args=[project.id]))
    ]
    grid = SQLFORM.grid(db.project, create=False, links=notes, fields=[db.project.name, db.project.employee_name, db.project.company_name, db.project.start_date, db.project.due_date, db.project.completed], deletable=False, maxtextlength=50)
    return locals()

@auth.requires_login()
def note():
    project = db.project(request.args(0))
    db.note.post_id.default = project.id
    form = crud.create(db.note) if auth.user else "Login to Post to the Project"
    allnotes = db(db.note.post_id==project.id).select()
    return locals()
```

Take a look. Add some notes. Now let's add a new view called `default/note.html`:

```
 {{extend 'layout.html'}}
<h2>Project Notes</h2>
<br/>
<h4>Current Notes</h4>
{{for n in allnotes:}}
<ul>
<li>{{=db.auth_user[n.created_by].first_name}} on {{=n.created_on.strftime
e("%m/%d/%Y")}}
- {{=n.post}}</li>
</ul>
{{pass}}
<h4>Add a note</h4>
{{=form}}<br>
```

We also need to give our application access to the web2py *crud* tools.

db.py

Change this:

```
from gluon.tools import Auth, Service, PluginManager

# host names must be a list of allowed host names (glob syntax allowed)
auth = Auth(db, host_names=myconf.get('host.names'))
service = Service()
plugins = PluginManager()
```

To this:

```
from gluon.tools import Auth, Service, PluginManager, Crud

# host names must be a list of allowed host names (glob syntax allowed)
auth = Auth(db, host_names=myconf.get('host.names'))
service = Service()
plugins = PluginManager()
crud = Crud(db)
```

Finally, let's update the `index()` function to add a button:

```
@auth.requires_login()
def index():
    response.flash = T('Welcome!')
    notes = [lambda project: A('Notes', _class="btn", _href=URL("default","note",args=[project.id]))]
    grid = SQLFORM.grid(db.project, create=False, links=notes,
                        fields=[db.project.name, db.project.employee_name, db.project.company_name,
                                db.project.start_date, db.project.due_date, db.project.completed], deletable=False,
                        maxtextlength=50)
    return locals()
```

NOTES

VIEW

EDIT

We just added the `btn` class to the `notes` variable.

Error Handling

web2py handles errors much differently than other frameworks. Tickets are automatically logged, and web2py does not differentiate between the development and production environments.

Have you seen an error yet? Remove the closing parenthesis from the following statement in the `index()` function: `response.flash = T('Welcome!')`. Now navigate to the homepage. You should see that a ticket number was logged. When you click on the ticket number, you get the specific details regarding the error.

You can also view all tickets here: <http://localhost:8000/admin/errors/py2manager>

You do not want users seeing errors, create `routes.py` in the root of the project (`web2py/`). Add the following code:

```
routes_onerror = [
    ('*/*', '/py2manager/static/error.html')
]
```

Then add the `error.html` file to the "static" directory:

```
<h2>This is an error. We are working on fixing it.</h2>
```

Kill (ctrl-c) and restart the server.

Refresh the homepage to see the new error message. Now errors are still logged, but end users won't see them. Correct the error.

Homework

- Please read the web2py [documentation](#) regarding error handling.

Interlude: Web Scraping and Crawling

Since data is not always accessible through a manageable format via web APIs, we sometimes need to get our hands dirty to access the data that we need. So, we need to turn to web scraping.

Web scraping is an automated means of retrieving data from a web page. Essentially, we grab unstructured HTML and parse it into usable data format that Python can work with. Most web page owners and many developers do not view scraping in the highest regard. The question of whether it's illegal or not often depends on *what* you do with the data, not the actual act of scraping. If you scrape data from a commercial website, for example, and resell that data, there could be serious legal ramifications. The scraping, if done ethically, is *generally* not illegal, if you use the data for your own personal use.

That said, most developers will tell you to follow these two principles:

1. Adhere to ethical scraping practices by not making repeated requests in a short period of time to a web server. Doing so will eat up bandwidth, which can slow down the site for other users and potentially overload the server.
2. Always check a website's acceptable use policy before scraping its data to see if accessing the website by using automated tools is a violation of its terms of use or service.

Example [terms of service](#) from Ebay, explicitly banning scraping:

content;

- take any action that may undermine the feedback or ratings systems (see [about our Feedback policies](#));
- transfer your eBay account (including Feedback) and user ID to another party without our consent;
- distribute or post spam, unsolicited or bulk electronic communications, chain letters, or pyramid schemes;
- distribute viruses or any other technologies that may harm eBay or the interests or property of users;
- use any robot, spider, scraper, data mining tools, data gathering and extraction tools, or other automated means to access our Services for any purpose, except with the prior express permission of eBay;
- interfere with the working of our Services, or impose an unreasonable or disproportionately large load on our infrastructure;
- export or re-export any eBay application or tool, except in compliance with the export control laws of any relevant jurisdictions and in accordance with posted rules and restrictions;
- infringe the copyright, trademark, patent, publicity, moral, database, and/or other intellectual property rights (collectively, "Intellectual Property Rights") that belong to or are licensed to eBay. Some, but not all, actions that may

It's absolutely vital to adhere to ethical scraping. You could very well get yourself banned from a website if you scrape millions of pages using a loop. With regard to the second principle, there is much debate about whether accepting a website's terms of use

is a binding contract or not. This is not a course on ethics or law, though. So, the examples covered will adhere to **both** principles.

Finally, it's also a good idea to check the *robots.txt* file before scraping or crawling. Usually found in the root directory of a web site, *robots.txt* establishes a set of rules that web crawlers or robots *should* adhere to.

Let's look at an example. Navigate to the HackerNews' *robots.txt* file:

<https://news.ycombinator.com/robots.txt>:

```
User-Agent: *
Disallow: /x?
Disallow: /vote?
Disallow: /reply?
Disallow: /submitted?
Disallow: /submitlink?
Disallow: /threads?
Crawl-delay: 30
```

- The User-Agent is the robot, or crawler, itself. Nine times out of ten you will see a wildcard, `*`, used as the argument, specifying that *robots.txt* applies to all robots.
- Disallow parameters establish the directories or files - "Disallow: /folder/" or "Disallow: /file.html" - that robots must avoid.
- The Crawl-delay parameter is used to indicate the minimum delay (in seconds) between successive server requests. So, in the HackerNews' example, after scraping the first page, a robot must wait thirty seconds before moving on to the next page and scraping it, and so on.

WARNING: Regardless of whether a Crawl-delay is established or not, it's good practice to wait a few seconds between each request to avoid putting unnecessary load on the server. Again, exercise caution. You do not want to get banned from a site.

And with that, let's start scraping...

Libraries

There are a number of great libraries you can use for extracting data from websites. If you are new to web scraping, start with [Beautiful Soup](#). It's easy to learn, simple to use, and the documentation is great. That being said, there are plenty of examples of using Beautiful Soup in the first Real Python [course](#). Start there. We're going to be looking at a more advanced library called [Scrapy](#).

Let's get Scrapy installed:

```
$ pip install scrapy==1.2.1
```

NOTE: If you are on a Windows machine, there are additional steps and dependencies that you need to install. Please follow this [video](#) for details. Just make sure you install the correct version of Scrapy - 1.2.1.

HackerNews (scrapy.Spider)

In this first example, let's scrape [HackerNews](#).

Once Scrapy is installed, open your terminal and create a new directory called "scraping", activate a new virtual environment, and then start a new Scrapy project:

```
$ scrapy startproject hackernews
New Scrapy project 'hackernews', using template directory '/usr/local/lib/python2.
7/site-packages/scrapy/templates/project', created in:
/Users/peterjeffryes/Documents/real-python/scrapy/hackernews

You can start your first spider with:
cd hackernews
scrapy genspider example example.com
```

This will create a "hackernews" directory with the following contents:

```
└── hackernews
    ├── __init__.py
    ├── items.py
    ├── pipelines.py
    ├── settings.py
    └── spiders
        └── __init__.py
└── scrapy.cfg
```

In this basic example, we only need to worry about the creation of the actual spider, which is the Python script used for scraping.

Start by creating a basic boilerplate:

```
$ cd hackernews
$ scrapy genspider basic news.ycombinator.com
```

Now we just need to customize this boilerplate.

Open up the *items.py* file in your text editor and edit it to define the fields that you want extracted. Let's grab the title and url from each posting on HackerNews:

```

import scrapy

class HackernewsItem(scrapy.Item):
    title = scrapy.Field()
    url = scrapy.Field()

```

Now, let's create the actual spider, which is already partially started for us in the *basic.py* file found in the "spiders" directory
("/scraping/hackernews/hackernews/spiders/basic.py"):

```

from scrapy import Spider
from scrapy.selector import Selector

from hackernews.items import HackernewsItem


class BasicSpider(Spider):
    # name the spider
    name = "basic"

    # allowed domains to scrape
    allowed_domains = ["news.ycombinator.com"]

    # urls the spider begins to crawl from
    start_urls = ['https://news.ycombinator.com/']

    # parses and returns the scraped data
    def parse(self, response):
        titles = Selector(response).xpath('//tr[@class="athing"]/td[3]')

        for title in titles:
            item = HackernewsItem()
            item['title'] = title.xpath("a[@href]/text()").extract()
            item['url'] = title.xpath("a/@href").extract()
            yield item

```

Navigate to the main directory ("/hackernews/hackernews") and run the following command: `scrapy crawl basic`. This will scrape all the data and output it to the screen. If you want to create a CSV so the parsed data is easier to read, run this command instead: `scrapy crawl basic -o items.csv -t csv`.

What's going on here?

Essentially, we used **XPath** to parse and extract the data using HTML tags:

1. `//tr[@class="athing"]/td[3]` - finds the third `<td>` after the element with `class="athing"`.
2. `a[@href]/text()` - finds all `<a>` tags within each `<td>` tag, then extracts the text
3. `a/@href` - again finds all `<a>` tags within each `<td>` tag, but this time it extracts the actual url

How did I know which HTML tags to use?

Open the start url in Chrome: <https://news.ycombinator.com/>. Right click on the first article link and select "Inspect Element". In the Chrome Developer Tools console, you can see the HTML that's used to display the first link:

```

<tr class="athing">
  <td align="right" valign="top" class="title">
    <span class="rank">1.</span>
  </td>
  <td valign="top" class="votelinks">
    <center>
      <a id="up_10606355" href="vote?for=10606355&dir=up&auth=2febc45204aa28f15f87c085df8fcfe96a99d85d&goto=news"><div class="votearrow" title="upvote"></div></a>
    </center>
  </td>
  <td class="title">
    <a href="https://mikeash.com/pyblog/friday-qa-2015-11-20-covariance-and-contravariance.html">Covariance and Contra
    variance</a>
    <span class="sitebit comhead"> (<a href="from?site=mikeash.com"><span class="site
    estr">mikeash.com</span></a>)</span>
  </td>
</tr>

```

You can see that everything we need, text and url, is located between the `<td class="title">` `</td>` tag:

```

<td class="title">
  <span class="deadmark"></span>
  <a href="https://mikeash.com/pyblog/friday-qa-2015-11-20-covariance-and-contravariance.html">Covariance and Contra
  variance</a>
  <span class="sitebit comhead"> (<a href="from?site=mikeash.com"><span class="site
  estr">mikeash.com</span></a>)</span>
</td>

```

And if you look at the rest of the document, all other postings fall within the same tag.

Thus, we have our main XPath:

```
titles = Selector(response).xpath('//tr[@class="athing"]/td[3]')
```

Now, we just need to establish the XPath for the title and url. Take a look at the HTML again:

```
<a href="https://mikeash.com/pyblog/  
friday-qa-2015-11-20-covariance-and-contravariance.html">Covariance and Contravari  
ance</a>
```

Both the title and url fall within the `<a>` tag. So our XPath must begin with those tags. Then we just need to extract the right attributes, `text` and `@href` respectively.

Need more help testing XPath expressions? Try the Scrapy Shell...

Scrapy Shell

Scrapy comes with an interactive tool called [Scrapy Shell](#) which easily tests XPath expressions. It's already included with the standard Scrapy installation.

The basic format for entering the shell is `scrapy shell <url>` :

```
$ scrapy shell http://news.ycombinator.com
```

Assuming there are no errors in the URL, you can now test your XPath expressions.

Start by using Developer Tools to get an idea of what to test. Based on the analysis we conducted a few lines up, we know that `//tr[@class="athing"]/td[3]` is part of the XPath used for extracting the title and URL. If you didn't know that, you could test it out in Scrapy Shell.

Type `sel.xpath('//tr[@class="athing"]/td[3]').extract()` in the Shell and press enter. It's hard to tell, but the URL and title are both part of the results. We're on the right track.

Add the `a` to the test:

```
sel.xpath('//tr[@class="athing"]/td[3]/a').extract()[0]
```

NOTE: By adding `[0]` to the end, we are just returning the first result.

Now you can see that just the title and URL are part of the results. Just extract the text and then the href:

```
sel.xpath('//tr[@class="athing"]/td[3]/a/text()').extract()[0]
```

and

```
sel.xpath('//tr[@class="athing"]/td[3]/a/@href').extract()[0]
```

Scrapy Shell is a valuable tool for testing whether your XPath expressions are targeting the correct data within the scraper. Try some more XPath expressions...

The "library" link at the bottom of the page:

```
sel.xpath('//span[@class="yclinks"]/a[3]/@href').extract()
```

The comment links and URLs:

```
sel.xpath('//td[@class="subtext"]/a/@href').extract()[0]
```

and

```
sel.xpath('//td[@class="subtext"]/a/text()').extract()[0]
```

See what else you can extract. Play around with this!

NOTE: If you need a quick primer on XPath, check out the W3C [tutorial](#). Scrapy also has some great [documentation](#). Also, before you start the next section, read [this](#) part of the Scrapy documentation. Make sure you understand the difference between the `scrapy.Spider` and `crawlSpider`.

Wikipedia (scrapy.Spider)

In this next example, we'll scrape a listing of movies from Wikipedia:

http://en.wikipedia.org/wiki/Category:2016_films

First, check the terms of use and the *robots.txt* file and answer the following questions:

- Does scraping or crawling violate their terms of use?
- Are we scraping a portion of the site that is explicitly disallowed?
- Is there an established crawl delay?

All no's, correct?

Start by building a scraper to scrape just the first page. Grab the movie title and URL. This is a slightly more advanced example than the previous one. Please try it on your own before looking at the code.

Start a new Scrapy project in the "scraping" directory:

```
$ scrapy startproject wikipedia
```

Create the *items.py* file:

```
import scrapy

class WikipediaItem(scrapy.Item):
    title = scrapy.Field()
    url = scrapy.Field()
```

Setup your crawler. Again, you can setup a skeleton crawler using the following command:

```
$ cd wikipedia
$ scrapy genspider wiki en.wikipedia.org
```

Finish coding the scraper:

```

from scrapy import Spider
from scrapy.selector import Selector

from wikipedia.items import WikipediaItem


class BasicSpider(Spider):
    # name the spider
    name = "wiki"

    # allowed domains to scrape
    allowed_domains = ["en.wikipedia.org"]

    # urls the spider begins to crawl from
    start_urls = ["http://en.wikipedia.org/wiki/Category:2016_films"]

    # parses and returns the scraped data
    def parse(self, response):

        titles = Selector(response).xpath('//div[@id="mw-pages"]/li')

        for title in titles:
            item = WikipediaItem()
            url = title.xpath("a/@href").extract()
            item["title"] = title.xpath("a/text()").extract()
            item["url"] = url[0]
            yield item

```

Save this to your "spiders" directory as `wiki.py`. Did you notice the XPath?

```
hxs.select('//div[@id="mw-pages"]/li')
```

This is equivalent to:

```
hxs.select('//div[@id="mw-pages"]/td/ul/li')
```

Since `` is a child element of `<div id="mw-pages">`, you can bypass the elements between them by using two forward slashes, `//`.

This time, output the data to a JSON file:

```
$ scrapy crawl wiki -o wiki.json -t json
```

Take a look at the results. We now need to change the relative URLs to absolute by appending `http://en.wikipedia.org` to the front of the URLs. First, import the `urlparse` library - `from urlparse import urljoin` - then update the `for` loop:

```
for title in titles:
    item = WikipediaItem()
    url = title.xpath("a/@href").extract()
    item["title"] = title.xpath("a/text()").extract()
    item["url"] = urljoin("http://en.wikipedia.org", url[0])
    yield item
```

Also, notice how there are some links without titles. These are not movies. We can easily eliminate them by adding a simple 'if' statement:

```
for title in titles:
    item = WikipediaItem()
    url = title.select("a/@href").extract()
    if url:
        item["title"] = title.select("a/text()").extract()
        item["url"] = urljoin("http://en.wikipedia.org", url[0])
        yield item
```

Your script should look like this:

```
from scrapy import Spider
from scrapy.selector import Selector
from urlparse import urljoin

from wikipedia.items import WikipediaItem


class BasicSpider(Spider):
    # name the spider
    name = "wiki"

    # allowed domains to scrape
    allowed_domains = ["en.wikipedia.org"]

    # urls the spider begins to crawl from
    start_urls = (
        "http://en.wikipedia.org/wiki/Category:2014_films",
    )

    # parses and returns the scraped data
    def parse(self, response):

        titles = Selector(response).xpath('//div[@id="mw-pages"]//li')

        for title in titles:
            item = WikipediaItem()
            url = title.select("a/@href").extract()
            if url:
                item["title"] = title.select("a/text()").extract()
                item["url"] = urljoin("http://en.wikipedia.org", url[0])
                yield item
```

Delete the JSON file and run the scraper again. You should now have the full URL.

Socrata (CrawlSpider and Item Pipeline)

In this next example, we'll scrape a listing of publicly available datasets from Socrata: <https://opendata.socrata.com/>.

Create the project:

```
$ scrapy startproject socrata
```

scrapy.Spider

Start with the basic spider (`scrapy.Spider`). We want the title, URL, and the number of views for each listing. Do this on your own.

items.py:

```
import scrapy

class SocrataItem(scrapy.Item):
    text = scrapy.Field()
    url = scrapy.Field()
    views = scrapy.Field()
```

Create the generic spider:

```
$ cd socrata
$ scrapy genspider opendata opendata.socrata.com
```

opendata.py:

```

from scrapy import Spider
from scrapy.selector import Selector

from socrata.items import SocrataItem


class OpendataSpider(Spider):
    name = "opendata"
    allowed_domains = ["opendata.socrata.com"]
    start_urls = ['https://opendata.socrata.com/']

    def parse(self, response):
        titles = Selector(response).xpath('//div[@itemscope="itemscope"]')
        for title in titles:
            item = SocrataItem()
            item["text"] = title.xpath('.//div[@class="browse2-result-title"]/h2/a/text()').extract()[0]
            item["url"] = title.xpath('.//div[@class="browse2-result-title"]/h2/a/@href').extract()[0]
            item["views"] = title.xpath('.//div[@class="browse2-result-view-count-value"]/text()').extract()[0].strip()
            yield item

```

Release the spider:

```
$ scrapy crawl opendata -o socrata.json
```

Make sure the JSON looks right.

CrawlSpider

Moving on, let's now look at how to crawl a website as well as scrape it. Basically, we'll start at the same starting URL, scrape the page, follow the first link in the pagination links at the bottom of the page. Then we'll start over on that page. Scrape. Crawl. Scrape. Crawl. Scrape. Etc.

Earlier, when you looked up the difference between the `scrapy.Spider` and `crawlSpider`, what did you find? Do you feel comfortable setting up the CrawlSpider? Give it a try.

First, there's no change to `items.py`. We will be scraping the same data on each page. Make a copy of `opendata.py`. Save it as `opendata_crawl.py`.

Update the imports:

```
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule

from socrata.items import SocrataItem
```

Update the name: `name = "opendatacrawl"` Add the rules just below the `start_urls`:

```
rules = [
    Rule(LinkExtractor(allow='browse\?utf8=%E2%9C%93&page=\d*'),
         callback='parse_item', follow=True)
]
```

What else do you have to update? First, the class must inherit from `CrawlSpider`, not `Spider`. Anything else?

```
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule

from socrata.items import SocrataItem

class OpendataSpider(CrawlSpider):
    name = "opendata_crawl"
    allowed_domains = ["opendata.socrata.com"]
    start_urls = ['https://opendata.socrata.com/']
    rules = [
        Rule(LinkExtractor(allow='browse\?utf8=%E2%9C%93&page=\d*'),
             callback='parse_item', follow=True)
    ]

    def parse_item(self, response):
        titles = Selector(response).xpath('//div[@itemscope="itemscope"]')
        for title in titles:
            item = SocrataItem()
            item["text"] = title.xpath('.//div[@class="browse2-result-title"]/h2/a/text()').extract()[0]
            item["url"] = title.xpath('.//div[@class="browse2-result-title"]/h2/a/@href').extract()[0]
            item["views"] = title.xpath('.//div[@class="browse2-result-view-count-value"]/text()').extract()[0].strip()
            yield item
```

As you can see, the only new parts of the code, besides the imports, are the rules, which define the crawling portion of the spider and the name of the parsing method:

```

rules = [
    Rule(LinkExtractor(allow='browse\\?utf8=%E2%9C%93&page=\\d*'),
         callback='parse_item', follow=True)
]

def parse_item(self, response):
    ...

```

NOTE: Please read over the [documentation](#) regarding rules quickly before you read the explanation. Also, it's important that you have a basic understanding of regular expressions. Please refer to the first [Real Python](#) course for a high-level overview.

So, the `LinkExtractor` is used to specify the links that should be crawled. The `allow` parameter is used to define the regular expressions that the URLs must match in order to be crawled.

Take a look at some of the URLs:

- <https://opendata.socrata.com/browse?utf8=%E2%9C%93&page=2>
- <https://opendata.socrata.com/browse?utf8=%E2%9C%93&page=3>
- <https://opendata.socrata.com/browse?utf8=%E2%9C%93&page=4>

What differs between them? The numbers on the end, right? So, we need to replace the number with an equivalent regular expression, which will recognize any number. The regular expression `\d` represents any number, 0 - 9. Then the `*` operator is used as a wildcard. Thus, any number will be followed, which will crawl every page in the pagination list.

We also need to escape the question mark (?) from the URL since question marks have special meaning in regular expressions. In other words, if we don't escape the question mark, it will be treated as a regular expression as well, which we don't want because it is part of the URL.

Thus, we are left with this regular expression:

```
browse\\?utf8=%E2%9C%93&page=\\d*
```

Make sense?

Remember how we said that we need to crawl "ethically"? Well, let's put a 10-second delay between each request.

WARNING: I cannot urge you enough to be **careful**. Only crawl sites where it is 100% legal at first. If you start venturing into gray area, do so at your own risk. These are powerful tools you are learning. Act responsibly. If you don't, getting banned from a site will be the least of your worries. Speaking of which, did you check the terms of use and the *robots.txt* file? If not, do so now.

To add a delay, open up the *settings.py* file, and then add the following code:

```
DOWNLOAD_DELAY = 10
```

Item Pipeline

Finally, instead of dumping the data into a JSON file, let's feed it to a database.

Create the database within the *first* "socrata" directory from your shell:

```
$ python
>>> import sqlite3
>>> conn = sqlite3.connect("project.db")
>>> cursor = conn.cursor()
>>> cursor.execute("CREATE TABLE data(text TEXT, url TEXT, views TEXT)")
<sqlite3.Cursor object at 0x1029db730>
```

Update the *pipelines.py* file:

```
import sqlite3

class SocrataPipeline(object):
    def __init__(self):
        self.conn = sqlite3.connect('project.db')
        self.cur = self.conn.cursor()

    def process_item(self, item, spider):
        self.cur.execute(
            "insert into data (text, url, views) values(?, ?, ?)",
            (item['text'], item['url'], item['views']))
        self.conn.commit()
        return item
```

Add the pipeline to the *settings.py* file:

```
ITEM_PIPELINES = {'socrata.pipelines.SocrataPipeline': 300}
```

Test this out with the BaseSpider first:

```
$ scrapy crawl opendata
```

Look good? Go ahead and delete the data using the SQLite Browser. Save the database.

Ready? Fire away:

```
$ scrapy crawl opendata_crawl
```

This will take a while. In the meantime, send a tweet about how awesome Real Python is! Is it still running? Take a break. Stretch. Once complete, open the database with the SQLite Browser. You should have about ~20,000 rows of data. Make sure to hold on to this data as we'll be using it later.

Homework

- Use your knowledge of Beautiful Soup, which, again, was taught in the first [Real Python](#) course, as well as the requests library, to scrape and parse all links from the web2py [homepage](#). Use a for loop to output the results to the screen. Refer back to the first course or the Beautiful Soup [documentation](#) for assistance.

Use the following command to install Beautiful Soup: `pip install beautifulsoup4`

NOTE: Want some more fun? We need web professional web scrapers.

Practice more with Scrapy. Make sure to upload everything to Github. Email us the link to info@realpython.com. We pay well.

Web Interaction

Web interaction and scraping go hand in hand. Sometimes, you need to fill out a form to access data or log in to a restricted area of a website. In such cases, Python makes it easy to interact in real-time with web pages. Whether you need to fill out a form, download a CSV file on a weekly basis, or extract a stock price each day when the stock market opens, Python can handle it. This basic web interaction combined with the data extracting methods we learned in the last lesson can create powerful tools.

Let's look at how to download a particular stock price:

```
# Download stock quotes in CSV

import requests
import time

i = 0

# obtain quote once every 3 seconds for the next 6 seconds
while (i < 2):

    # define the base url
    base_url = 'http://download.finance.yahoo.com/d/quotes.csv'

    # retrieve data from web server
    data = requests.get(
        base_url,
        params={'s': 'GOOG', 'f': 'sl1d1t1c1ohgv', 'e': '.csv'})

    # write the data to csv
    with open('stocks.csv', 'wb') as code:
        code.write(data.content)

    i += 1

    # pause for 3 seconds
    time.sleep(3)
```

Save this file as *stock_download.py* in the "scraping" directory and run it. Then load up the CSV file after the program ends to see the stock prices. You could change the sleep time to 60 seconds so it pulls the stock price every minute or 360 to pull it every hour. Just leave it running in the background.

Let's look at how we got the parameters: `params={'s': 'GOOG', 'f': 'sl1d1t1c1ohgv', 'e': '.csv'})`

Here is a link to download the spreadsheet:

<http://download.finance.yahoo.com/d/quotes.csv?s=goog&f=sl1d1t1c1ohgv&e=.csv>

So to download the CSV, we need to input parameters for `s`, `f`, and `e`, which you can see in the above URL. The parameters for `f` and `e` are constant, which means you could include them in the `base_url`. So it's just the actual stock quote that changes.

How would you then pull prices for a number of quotes using a loop? Think about this before you look at the answer.

```
# Download stock quotes in CSV

import requests
import time

i = 0

# define the stocks to download
stock_list = ['GOOG', 'YHOO', 'MSF']

while (i < 1):

    # define the base url
    base_url = 'http://download.finance.yahoo.com/d/quotes.csv'

    # retrieve data from web server
    for stock in stock_list:
        data = requests.get(
            base_url,
            params={'s': stock, 'f': 'sl1d1t1c1ohgv', 'e': '.csv'})
    )

    # output the data to the screen
    print(data.content)

    i += 1

    # pause for 3 seconds
    time.sleep(3)
```

web2py: REST Redux

Remember the data we scraped from Socrata? No? Go back and quickly review the lesson. Then, locate the *project.db* file on your local computer we used to store the data. We're going to build our own RESTful web service to expose the data that we scraped. Why would we want to do this when the data is already available?

1. The data could be in high demand but the Socrata website is unreliable. By scraping the data and providing it via a RESTful API, you can ensure the data is always available to you or your clients.
2. Again, the data could be in high demand but it's poorly organized. You can cleanse the data after scraping and offer it in a more human and machine readable format.
3. You want to create a [mashup](#). You could scrape data from other sites and combine that data with the data from Socrata and create an API.

Whatever the reason, let's look at how to quickly set up a RESTful web service via web2py by to expose the data we pulled from Socrata.

Remember:

- Each resource or endpoint should be identified by a separate URL.
- There are four HTTP methods used for interacting with RESTful APIs - GET, POST, PUT, and DELETE

Let's start with a basic example before using the scraped data from Socrata.

Basic REST

Remember how to set up a web2py project?

1. Create a new directory called "web2py-rest" to house your app.
2. Download the source code from the web2py [repository](#). Unzip the file, and place all files and folders into the "web2py-rest" directory.
3. Change directories into `web2py-rest`.
4. Using `pyenv`, set the local version of Python: `pyenv local 2.7.6`
5. Create a new app from your terminal: `pyenv exec python web2py.py -S basic_rest`. After this project is created, we are left in the web2py Shell. Go ahead and exit it.
6. Within the terminal, navigate to the "applications" directory, then into the "basic_rest" directory.

Now we can set up the API...

API

Create a new database table in in `db.py`:

```
db.define_table('fam',Field('role'),Field('name'))
```

Within the web2py admin, enter some dummy data into the newly created table. Add the RESTful functions to the [default.py controller](#)):

```
@request.restful()
def api():
    response.view = 'generic.'+request.extension
    def GET(*args, **vars):
        patterns = 'auto'
        parser = db.parse_as_rest(patterns,args,vars)
        if parser.status == 200:
            return dict(content=parser.response)
        else:
            raise HTTP(parser.status,parser.error)
    def POST(table_name,**vars):
        return db[table_name].validate_and_insert(**vars)
    def PUT(table_name,record_id,**vars):
        return db(db[table_name]._id==record_id).update(**vars)
    def DELETE(table_name,record_id):
        return db(db[table_name]._id==record_id).delete()
    return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)
```

These functions expose any field in the database to the outside world. If you want to limit the resources exposed, you'll need to define various [patterns](#). For example:

```
def GET(*args, **vars):
    patterns = [
        "/test[fam]",
        "/test/{fam.name.startswith}",
        "/test/{fam.name}/:field",
    ]
    parser = db.parse_as_rest(patterns,args,vars)
    if parser.status == 200:
        return dict(content=parser.response)
    else:
        raise HTTP(parser.status,parser.error)
```

These patterns define the following rules:

- http://127.0.0.1:8000/basic_rest/default/api/test -> GET all data
- http://127.0.0.1:8000/basic_rest/default/api/test/t -> GET a single data point where the "name" starts with "t"
- http://127.0.0.1:8000/basic_rest/default/api/test/1 -> Can you guess what this does?
Go back and look at the database schema for help.

For simplicity, let's expose everything.

Sanity Check

Test out the following GET requests in your browser:

http://127.0.0.1:8000/basic_rest/default/api/fam.json

```
{"content": [{"role": "Father", "id": 1, "name": "Tom"}, {"role": "Mother", "id": 2, "name": "Jane"}, {"role": "Brother", "id": 3, "name": "Jeff"}, {"role": "Sister", "id": 4, "name": "Becky"}]}
```

http://127.0.0.1:8000/basic_rest/default/api/fam/id/1.json

```
{"content": [{"role": "Father", "id": 1, "name": "Tom"}]}
```

Test out the following requests in the Shell and look at the results in the database:

```
>>> import requests
>>>
>>> payload = {"name" : "john", "role" : "brother"}
>>> r = requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json", data=payload)
>>> r
<Response [200]>
>>>
>>> r = requests.delete("http://127.0.0.1:8000/basic_rest/default/api/fam/2.json")
>>> r
<Response [200]>
>>>
>>> payload = {"name" : "Jeffrey"}
>>> r = requests.put("http://127.0.0.1:8000/basic_rest/default/api/fam/3.json", data=payload)
>>> r
<Response [200]>
```

Auth

In most cases, you do not want just anybody having access to your API like this. Besides, limiting the data points as described above, you also want to have user authentication in place.

Register a new user at http://127.0.0.1:8000/basic_rest/default/user/register, and then update the function in the controller, adding a login required decorator:

```
auth.settings.allow_basic_login = True

@auth.requires_login()
@request.restful()
def api():
    response.view = 'generic.'+request.extension
    def GET(*args, **vars):
        patterns = 'auto'
        parser = db.parse_as_rest(patterns,args,vars)
        if parser.status == 200:
            return dict(content=parser.response)
        else:
            raise HTTP(parser.status,parser.error)
    def POST(table_name,**vars):
        return db[table_name].validate_and_insert(**vars)
    def PUT(table_name,record_id,**vars):
        return db(db[table_name]._id==record_id).update(**vars)
    def DELETE(table_name,record_id):
        return db(db[table_name]._id==record_id).delete()
    return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)
```

Now you need to be authenticated to make any requests:

```
>>> import requests
>>> from requests.auth import HTTPBasicAuth
>>> payload = {"name" : "Katie", "role" : "Cousin"}
>>> auth = HTTPBasicAuth("Michael", "reallyWRONG")
>>> r = requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json", data=payload, auth=auth)
>>> r
<Response [403]>
>>>
>>> auth = HTTPBasicAuth("michael@realpython.com", "admin")
>>> r = requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json", data=payload, auth=auth)
>>> r
<Response [200]>
```

Test different endpoints before moving on.

Homework

- Watch [this](#) short video on REST.

Advanced REST

Now that you've seen the basics of creating a RESTful web service, let's build a more advanced example using the Socrata data.

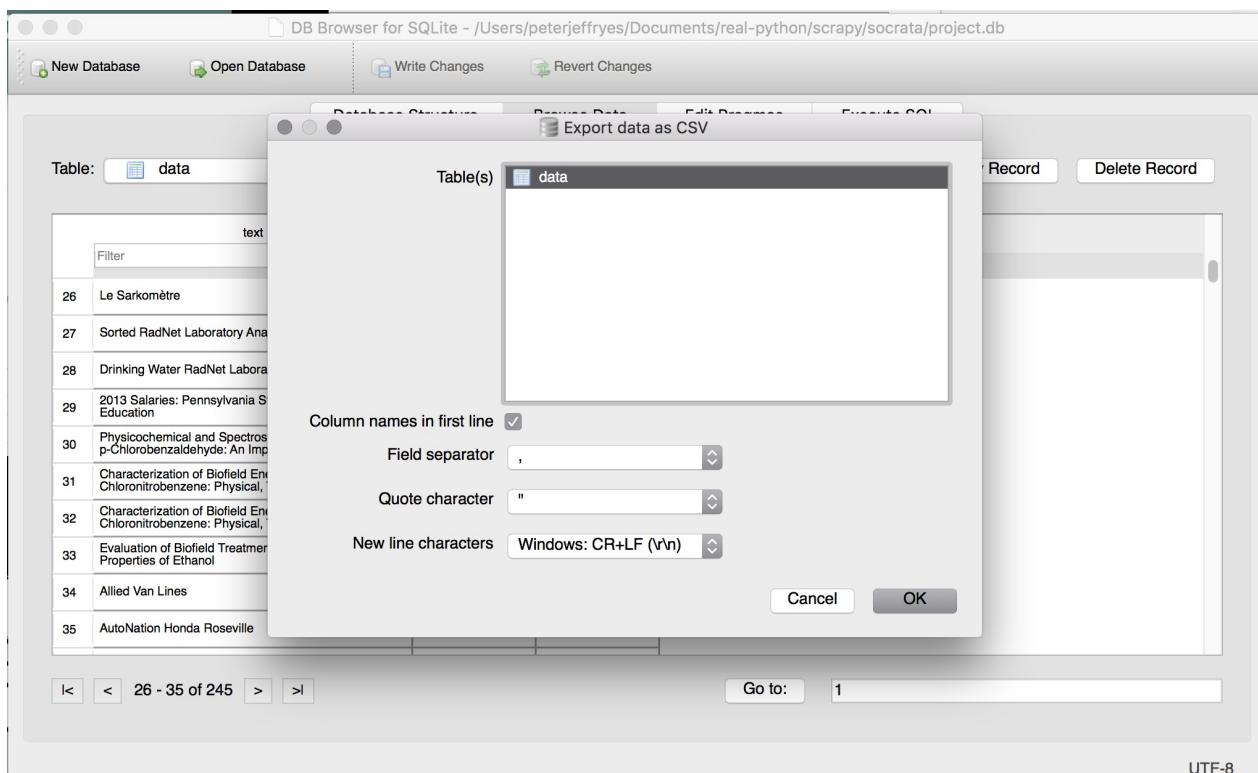
Setup

First, create a new app called "socrata" within "web2py-rest", then register a new user - <http://127.0.0.1:8000/socrata/default/user/register>, and then create a new table with the following schema:

```
db.define_table('socrata', Field('name'), Field('url'), Field('views'))
```

Now we need to extract the data from the *projects.db* and import it to the new database table you just created. There are a number of different ways to handle [this](#). We'll export the data from the old database in CSV format and then import it directly into the new web2py table.

Open *projects.db* in your SQLite Browser. Then click "File" -> "Export" -> "Table" as CSV file. Save the file in the following directory as *socrata.csv*: "../web2py-rest/applications/socrata".



NOTE: Be sure to use the settings for the export that are shown in the picture above. This will create a new line for each row as well as include the field names.

You need to rename the "text" field since it's technically a restricted name. Change this to "name".

To upload the CSV file, return to the **Edit** page on web2py, click the button for "database administration", then click the "db.socrata" link. Scroll to the bottom of the page and click "choose file" select *socrata.csv*. Now click import.

There should now be ~20,000 rows of data in the "socrata" table:

NOTE: In the future, when you set up your Scrapy Items Pipeline, you can dump the data right to the web2py database. The process is the same as outlined. Again, make sure to only grab the view count, not the word "views".

API Design

When designing your RESTful API, you should follow these [best practices](#):

- Keep it simple and intuitive
- Use HTTP methods
- Provide HTTP status codes
- Use simple URLs for accessing endpoints
- JSON should be the format of choice
- Use only lowercase characters

Add the following code to *default.py*:

```

@request.restful()
def api():
    response.view = 'generic.json'+request.extension
    def GET(*args, **vars):
        patterns = 'auto'
        parser = db.parse_as_rest(patterns,args,vars)
        if parser.status == 200:
            return dict(content=parser.response)
        else:
            raise HTTP(parser.status,parser.error)
    def POST(table_name,**vars):
        return db[table_name].validate_and_insert(**vars)
    def PUT(table_name,record_id,**vars):
        return db(db[table_name]._id==record_id).update(**vars)
    def DELETE(table_name,record_id):
        return db(db[table_name]._id==record_id).delete()
    return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)

```

Sanity Check

GET

Navigate to the following URL to see the endpoints that are available via GET -
<http://127.0.0.1:8000/socrata/default/api/patterns.json>

Output:

```
{
  content: [
    "/socrata[socrata]",
    "/socrata/id/{socrata.id}",
    "/socrata/id/{socrata.id}/:field"
  ]
}
```

Endpoints and patterns:

- <http://127.0.0.1:8000/socrata/default/api/socrata.json>
- <http://127.0.0.1:8000/socrata/default/api/socrata/id/ID.json>
- http://127.0.0.1:8000/socrata/default/api/socrata/id/ID/FIELD_NAME.json

Let's look at each one in detail with the Python Shell:

```
>>> import requests
```

<http://127.0.0.1:8000/socrata/default/api/socrata/id/ID.json>

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/
    default/api/socrata/id/1.json")
>>> r
<Response [200]>
>>> r.content
'{"content": [{"name": "Drug and Alcohol Treatments Florida",
    "views": "6", "url": "https://opendata.socrata.com/
    dataset/Drug-and-Alcohol-Treatments-Florida/u7mv-9jrm", "id": 100}]]}\n'
```

http://127.0.0.1:8000/socrata/default/api/socrata/id/ID/FIELD_NAME.json

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/
    default/api/socrata/id/100/name.json")
>>> r
<Response [200]>
>>> r.content
'{"content": [{"name": "Drug and Alcohol Treatments Florida"}]}\n'
>>> r = requests.get("http://127.0.0.1:8000/
    socrata/default/api/socrata/id/100/views.json")
>>> r
<Response [200]>
>>> r.content
'{"content": [{"views": "6"}]}\n'
```

POST

<http://127.0.0.1:8000/socrata/default/api/socrata.json>

```
>>> payload = {'name': 'new database', 'url': 'http://new.com', 'views': '22'}
>>> r = requests.post("http://127.0.0.1:8000/
    socrata/default/api/socrata.json", payload)
>>> r
<Response [200]>
```

PUT

<http://127.0.0.1:8000/socrata/default/api/socrata/ID.json>

```

>>> payload = {'name': 'new database'}
>>> r = requests.put("http://127.0.0.1:8000/
    socrata/default/api/socrata/3.json", payload)
>>> r
<Response [200]>
>>> r = requests.get("http://127.0.0.1:8000/
    socrata/default/api/socrata/id/3/name.json")
>>> r
<Response [200]>
>>> r.content
'{"content": [{"name": "new database"}]}\\n'

```

DELETE

<http://127.0.0.1:8000/socrata/default/api/socrata/ID.json>

```

>>> r = requests.delete("http://127.0.0.1:8000/
    socrata/default/api/socrata/3.json")
>>> r
<Response [200]>
>>> r = requests.get("http://127.0.0.1:8000/
    socrata/default/api/socrata/id/3/name.json")
>>> r
<Response [404]>
>>> r.content
'no record found'

```

Authentication

Finally, make sure to add the login required decorator to the `api()` function, so that users have to be registered to make API calls:

```

auth.settings.allow_basic_login = True
@auth.requires_login()

```

Authorized:

```

>>> r = requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/1.json"
, auth=('michael@realpython.com', 'admin'))
>>> r.content
'{"content": [{"name": "2010 Report to Congress on
    White House Staff", "views": "508,705",
    "url": "https://opendata.socrata.com/Government/
    2010-Report-to-Congress-on-White-House-Staff/vedg-c5sb", "id": 1}]}\\n'

```


Django: Quick Start

[Django](#), like [web2py](#), is a high-level web framework, used for rapid web development. With a strong community of supporters and some of the largest, most popular [sites](#) using it - such as [Reddit](#), [Instagram](#), [Mozilla](#), [Pinterest](#), [Disqus](#), and [Rdio](#), to name a few - it's the most well-known and used Python web framework.



WARNING: Django has a high learning curve due to much of the implicit automation that happens behind the scenes. It's much more important to understand the basics - e.g., the Python syntax and language, web client and server fundamentals, request/response cycle, etc. - and then move on to one of lighter frameworks (like [Flask](#) or [bottle.py](#)) so that when you do start developing with Django, it will be much easier to obtain a deeper understanding of the automation that happens and its integrated functionality. Even [web2py](#), which is slightly more automated, is easier to learn because it was specifically designed as a learning tool.

Django projects are logically organized around the Model-View-Controller (MVC) architecture. However, Django's MVC flavor is slightly different in that the views act as the controllers. So, projects are actually organized in a [Model-Template-View](#):

- **Models** represent your data model, traditionally in the form of a relational database. Django uses the Django ORM to organize and manage databases, which functions in relatively the same manner, despite a much different syntax, as SQLAlchemy and [web2py](#)'s DAL.
- **Templates** visually represent the data model (like the views in the MVC architecture). This is the presentation layer and defines how information is displayed to the end user.

- **Views** define the business logic (like the controllers in the MVC architecture), which logically link the templates and models.

This can be a bit confusing, but just [remember](#) that the MTV and MVC architectures work *relatively* the same.

In this chapter you'll see how easy it is to get a project up and running due to the automation of common web development tasks and integrated functionality. As long as you are aware of the inherent structure and organization that Django uses, you can focus less on monotonous tasks, inherent in web development, and more on developing the higher-level portions of your application.

Brief History

Django grew organically from a development team at the [Lawrence Journal-World](#) newspaper in Lawrence, Kansas (home of the University of Kansas) in 2003. The developers realized that web development up to that point followed a similar pattern resulting in much [redundancy](#):

1. Write a web application from scratch
2. Write another web application from scratch
3. Realize the first application shares much in common with the second application
4. Refactor the code so that the first application shares code with the second application
5. Repeat steps 2 - 4 several times
6. Realize you've invented a framework

The developers found that the commonalities shared between most applications could (and should) be automated. Django came to fruition from this rather simple realization, changing the state of web development as a whole.

Homework

- Read Django's [Design Philosophies](#)
- Optional: To gain a deeper understanding of the history of Django and the current state of web development, read the [Introducing Django](#) of the [Django Book](#).

Installation

Make a new directory called "django-quick-start", and then create and activate a virtual environment.

Install Django:

```
$ pip install django==1.10.3
```

NOTE: Because we will be using Python 3 in this chapter, we are going to go back to *virtual environment* for our virtual environment. Feel free to use whichever virtual environment tool you prefer.

Want to check the Django version currently installed within your virtual environment? Open the Python shell and run the following commands:

```
>>> import django
>>> django.VERSION
(1, 10, 3, 'final', 0)
```

If you see a different version or an `ImportError`, make sure to uninstall Django, `pip uninstall Django`, and then try installing Django again. Double-check that your virtual environment is activated before installing.

Hello, World!

As always, let's start with a basic "Hello World" app.

Basic Setup

Start a new Django project:

```
$ django-admin.py startproject hello_world_project
```

This command created the basic project layout, containing one directory and five files:

```
└── hello_world_project
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
    └── manage.py
```

NOTE: If you just type `django-admin.py` , you'll be taken to the help section, which displays all the available commands that can be performed with `django-admin.py` . Use this if you forget the name of a command.

For now you just need to worry about the `manage.py`, `settings.py`, and `urls.py` files:

- **`manage.py`:** This file is a command-line utility, used to manage and interact with the Django project and database. You probably won't ever have to edit the file itself; however, it is used with almost every process as you develop your project. Run `python manage.py help` to learn more about all the commands supported by `manage.py`.
- **`settings.py`:** This is the settings file for your project, where you define your project's configuration settings, such as database connections, external applications, template files, and so forth. There are numerous defaults setup in this file, which often change as you develop and deploy your Project.
- **`urls.py`:** This file contains the URL mappings, connecting URLs to Views.

Before we start creating our project, let's make sure everything is setup correctly by running the built-in development server. Navigate into the first "hello_world_project" directory and run the following command:

```
$ python manage.py runserver
```

NOTE: Don't worry about the migration warning that you get. Keep going!

You can specify a different port with the following command (if necessary):

```
$ python manage.py runserver 8080
```

You should see something similar to this:

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

December 03, 2016 - 21:55:53
Django version 1.10.3, using settings 'hello_world_project.settings'
Starting development server at http://127.0.0.1:8080/
Quit the server with CONTROL-C.
```

Take note of *You have unapplied migrations; your app may not work properly until they are applied. Run 'python manage.py migrate' to apply them.* Before we can test the project, we need to apply the database migrations. We'll discuss this in the next chapter. For now, kill the development server by pressing Control-C within the terminal, then run the command:

```
$ python manage.py migrate
```

Run the server again:

```
$ python manage.py runserver
```

Navigate to <http://localhost:8000/> to view the development server:



Exit the development server.

Adding a Project to Sublime Text

It's much easier to work with Sublime Text by adding all the Django files to a Sublime Project.

- Navigate to "Project" -> "Add folder to Project"
- Find the "django-quick-start" directory then click "Open"
- Save the project for easy access by navigating again to "Project" -> "Save Project" and save as *hello-world-project.sublime-project* within your "django" directory.

You should see both directories and all files on the left pane. Now if you exit out of Sublime and want to start working on the same project, you can just go to "Project" -> "Open Project", and then find the saved *hello-world-project.sublime-project* file.

Create a new App

Now that the Django project is setup, let's create a new app. With virtual environment activated run the following command:

```
$ python manage.py startapp hello_world
```

This will create a new directory called "hello_world", which includes the following files:

- *admin.py*: Used to register an app's models with the Django admin panel
- *models.py*: Used to define your data models and map them to database table
- *tests.py*: Houses your test code used for testing your application (don't worry about this for now)
- *views.py*: This file is your application's controller, defining the business logic in order to accept HTTP requests and return responses back to the user

Your project structure should now look like this:

```
└── db.sqlite3
└── hello_world
    ├── __init__.py
    ├── admin.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── hello_world_project
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py
```

NOTE: *What's the difference between a Django project and an App?* A project is the main web app, containing the settings, templates, and URL routes for a set of Django Apps. Meanwhile, a Django app is just an application that has an individual function such as a blog or message forum. Each app should have a *separate* function associated with it, distinct from other Apps. Django Apps are used to encapsulate common functions. Put another way, the project is your final product (your entire web app), comprised of separate functions from each app (each individual feature of your App, like - user authentication, a blog, registration, and so on).

By breaking Projects into a series of small Apps, you can theoretically reuse a Django app in a different Django project - so there's no need to reinvent the wheel.

Next, we need to include the new app in the *settings.py* file so that Django knows that it exists. Scroll down to "INSTALLED_APPS" and add the app name, *hello_world*, to the end of the tuple:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'hello_world',
)
```

URLs and Views

Open the `views.py` file and add the following code:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse('<html><body>Hello, World!</body></html>')
```

What's going on here?

- This function, `index()`, takes a parameter, `request`, which is an object that has information about the HTTP request.
- We named the function `index()` but as you will see in a second, this doesn't matter - you can name the function whatever you wish. By convention, though, make sure the function name represents the main objective of the function itself.
- A response object is then instantiated, which returns the text `<html><body>Hello, World!</body></html>` to the browser.

Add a `urls.py` file to the the "hello_world" app, then add the following code to link (or map) a URL to our specific `home` view:

```
from django.conf.urls import url
from hello_world import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

With our app's Views and URLs setup, we now just need to link the project URLs to the app URLs. To do this, add the following code to the project `urls.py` in the "hello_world_project" folder:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/', include('hello_world.urls')),
]
```

SEE ALSO: Please read the official Django documentation on [URLs](#) for further details and examples.

Let's test it out. Fire up the server - `python manage.py runserver`, and then open your browser to <http://localhost:8000/hello>. It worked! You should see the "Hello, World!" text in the browser.



Navigate back to the root - <http://localhost:8000/>. You should see a 404 error since we have not defined a view that maps to `/`. Let's change that.

Open up your Project's `urls.py` file again and update the `urlpatterns` list:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/', include('hello_world.urls')),
    url(r'^$', include('hello_world.urls')),
]
```

Save the file and refresh the page. You should see the same "Hello, World!" text. So, we simply assigned or mapped two URLs (`/` and `/hello`) to that single view.

Remove the pattern, `url(r'^hello/', include('hello_world.urls'))`, so that we can just access the page through the `/` url.

Homework

Experiment with adding additional text-based views within the app's `urls.py` file and assigning them to URLs. For example:

view:

```
def about(request):
    return HttpResponse(
        "Here is the About Page. Want to return home? <a href='/'>Back Home</a>"
    )
```

url:

```
url(r'^about/', views.about, name='about'),
```

Templates

Django templates are similar to web2py's views, which are used for displaying HTML to the user. You can also embed Python code directly into the templates using template tags. Let's modify the example above to take advantage of this feature.

Navigate to the project root and create a new directory called "templates". Your project structure should now look like this:

```
└── hello_world
    ├── __init__.py
    ├── admin.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    ├── urls.py
    └── views.py
└── hello_world_project
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py
└── templates
```

Next, update the `settings.py` file to add the path to the "templates" directory to the list associated with the `DIRS` key so that your Django project knows where to find the templates:

```
'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

Update the `views.py` file:

```
from django.template import Context, loader
from datetime import datetime
from django.http import HttpResponseRedirect

def index(request):
    return HttpResponseRedirect('<html><body>Hello, World!</body></html>')

def about(request):
    return HttpResponseRedirect(
        "Here is the About Page. Want to return home? <a href='/'>Back Home</a>")
    )

def better(request):
    t = loader.get_template('betterhello.html')
    c = Context({'current_time': datetime.now(), })
    return HttpResponseRedirect(t.render(c))
```

So, in the `better()` function we used the `loader.get_template()` to return a template called `betterhello.html`. The `Context` class takes a dictionary of variable names, as the dict's keys, and associated values, as the dict's values (which, again, is similar to the web2py syntax we saw earlier). Finally, the `render()` function returns the `Context` variables to the template.

Next, create a new HTML file called `betterhello.html` in the "template" directory and pass in the key from the dictionary surrounded by double curly braces `{} {}` :

```
<html>
<head><title>A Better Hello!</title></head>
<body>
  <p>Hello, World! This template was rendered on {{current_time}}.</p>
</body>
</html>
```

Finally, we need to update the `urls.py` file:

```
url(r'^better/$', views.better, name='better'),
```

Fire up your server and navigate to <http://localhost:8000/better>. You should see something like this:



Here, we have a placeholder in our view, `{{current_time}}`, which is replaced with the current date and time from the views, `{'current_time': datetime.now(),}`. If you see an error, double check your code. One of the most common errors is that the templates directory path is incorrect in your `settings.py` file. Try adding a `print` statement to double check that path is correct to the `settings.py` file - `print(os.path.join(BASE_DIR, 'templates'))`.

Notice the time. Is it correct? The time zone defaults to UTC, `TIME_ZONE = 'UTC'`. If you would like to change it, open the `settings.py` file, and then change the COUNTRY/CITY based on the timezones found in [Wikipedia](#).

For example, if you change the time zone to `TIME_ZONE = 'America/Denver'`, and refresh <http://localhost:8000/better>, it should display the U.S. Mountain Standard Time.

Change the time zone so that the time is correct based on your location.

Workflow

Before moving on, let's take a look at the basic workflow used for creating a Django project and App...

Create a Project

1. Run `python django-admin.py startproject <name_of_the_project>` to create a new Project. This will create the project in a new directory. Enter that new directory.
2. Migrate the database via `python manage.py migrate`.

Creating an App

1. Run `python manage.py startapp <name_of_the_app>`.
2. Add the name of the app to the `INSTALLED_APPS` tuple within `settings.py` file so that Django knows that the new app exists.
3. Link the Application URLs to the main URLs within the Project's `urls.py` file.
4. Add the views to the Application's `views.py` file.
5. Add a `urls.py` file within the new application's directory to map the views to specific URLs.
6. Create a "templates" directory, update the template path in the "settings.py" file, and finally add any templates.

Interlude: Introduction to JavaScript and jQuery

Before moving to the next Django project, let's finish the front-end tutorial by looking at part two - **Introduction to JavaScript and jQuery**.

NOTE: Did you miss part 1? Jump back to [Interlude: Introduction to HTML and CSS](#).

Getting Started

Start by adding a `main.js` file to the root directory and include the following code:

```
$(function() {
  console.log("whee!")
});
```

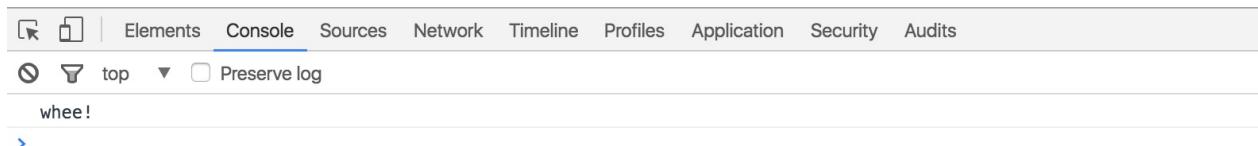
Then add the following files to your `index.html` just before the closing `</body>` tag:

```
<script src="http://code.jquery.com/jquery-2.2.4.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/
  bootstrap/3.3.7/js/bootstrap.min.js"></script>
<script src="main.js"></script>
```

Here we are just including the jQuery and Bootstrap dependencies as well as and our custom JavaScript file, `main.js`.

Open the "index.html" file in your web browser. In the JavaScript file there is a `console.log`. This is a debugging tool that allows you to post a message to the browser's JavaScript console found with Chrome's Developer Tools.

Open your console. You should see the text "whee!".



Now, insert a word into the input box and click Submit. Nothing happens. We need to somehow grab that inputted word and do *something* with it.

Handling the Event

The process of grabbing the inputted word from the form when a user hits "Submit" is commonly referred to as an [event handler](#). In this case, the event is the actual button click. We will use jQuery to "handle" that event.

NOTE: Please note that jQuery is Javascript, just a set of helper methods developed in JavaScript. It is possible to perform the exact same functionality jQuery provides in vanilla JavaScript - it just takes more code.

Update *main.js*:

```
$(function() {  
  
  console.log("whee!")  
  
  // event handler  
  $("#btn-click").click(function() {  
    if ($('#input').val() !== '') {  
      var input = $("input").val()  
      console.log(input)  
    }  
  });  
  
});
```

Add `id="btn-click"` to "index.html" within the `<button>` tags:

```
<button id="btn-click" class="btn btn-primary btn-md">Submit!</button>
```

What's going on?

1. `$("#btn-click").click(function() {` is the event. This initiates the process, running the code in the remainder of the function. In other words, the remaining JavaScript will not run until there is a button click.
2. `var input = $("input").val()` sets a variable called `input` with the inputted value from the form, which is grabbed via `.val()`.
3. `id="btn-click"` is used to tie the HTML to the JavaScript. This `id` is referenced in the initial event within the JavaScript file - `"#btn-click"`.
4. `console.log(input)` displays the word to the end user via the JavaScript console.

NOTE: The `$()` in `$("#btn-click").click()` is a jQuery constructor. Basically, it's used to tell the browser that the code within the parenthesis is jQuery.

Open "index.html" in your browser. Make sure you have your JavaScript console open. Enter a word in the input box and click the button. This should display the word in the console



Todos

Get yourself organized!

Real Python

Submit!



Updating the DOM

Next, instead of using a `console.log` to display the inputted word to the user, let's add it to the [Document Object Model](#) (DOM). Wait. What's the DOM? Put simply, the DOM is a structural representation of the HTML document. Using JavaScript, you can add, remove, or modify the contents of the DOM, which changes how the page looks to the end user.

Open up your JavaScript file and add this line of code:

```
$( 'ol' ).append( '<li><a href="">x</a>' + input + '</li>' );
```

Updated file:

```
$(function() {  
  
  console.log("whee!")  
  
  // event handler  
  $("#btn-click").click(function() {  
    if ($("#input").val() !== '') {  
      // grab the value from the input box after the button click  
      var input = $("#input").val()  
      // display value within the browser's JS console  
      console.log(input)  
      // add the value to the DOM  
      $('ol').append('<li><a href="">x</a> - ' + input + '</li>');  
    }  
    $("#input").val('');  
  });  
});
```

Then add the following code to your `index.html` file, just below the button:

```
<br>  
<br>  
<h2>Todos</h2>  
<h3>  
  <ol class="results"></ol>  
<h3>
```

What's going on?

`($('ol').append('x' + input + '');` adds the value of the `input` variable to the DOM between the `<ol class="results">` and ``. Further, we're adding the value from the input plus some HTML, `x - `.
`($('input').val(''));` clears the input box.

Test this out in your browser.

Before moving on, we need to make one last update to `main.js` to remove todos from the DOM once complete.

Add the following code to `main.js`

```
$document.on('click', 'a', function (event) {
  event.preventDefault();
  $(this).parent().remove();
});
```

The final code:

```
$(function() {
  console.log("whee!")

  // event handler
  $("#btn-click").click(function() {
    if ($('input').val() !== '') {
      // grab the value from the input box after the button click
      var input = $("input").val()
      // display value within the browser's JS console
      console.log(input)
      // add the value to the DOM
      $('ol').append('<li><a href="">x</a> - ' + input + '</li>');
    }
    $('input').val('');
  });

  $document.on('click', 'a', function (event) {
    event.preventDefault();
    $(this).parent().remove();
  });
})
```

Here, on the event, the click of the link, we're removing that specific todo from the DOM. `event.preventDefault()` cancels the default action of the click, which is to follow the link. Try removing this to see what happens. `this` refers to the current object, `a`, and we're removing the parent element, ``.

Test this out.

Homework

- Ready to test your skills? Check out [Practicing jQuery with the Simpsons](#) along with the other front-end [resources](#).
- Go through the Codecademy tracks on [JavaScript](#) and [jQuery](#) for more practice.
- Spend some time with Chrome's Dev Tools. Read the official [documentation](#) for help.

Django Bloggy: A blog app (part one)

In this next project we'll develop a simple blog application with Django using the workflow introduced at the end of the *Django: Quick Start* chapter.

Setup

Setup a new Django project:

```
$ mkdir django-bloggy
$ cd django-bloggy
$ python3 -m venv env
$ source env/bin/activate
$ pip install django==1.10.3
$ pip freeze > requirements.txt
$ django-admin.py startproject bloggy_project
```

Add the Django project folder to your text editor.

Model

Database setup

Open up `settings.py` and navigate to the `DATABASES` dict. Notice that SQLite is the default database. Let's change the name to `bloggy.db`:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': os.path.join(BASE_DIR, 'bloggy.db'),  
    }  
}
```

Sync the database

```
$ cd bloggy_project  
$ python manage.py migrate
```

This command, `migrate`, creates the basic tables based on the apps in the `INSTALLED_APPS` tuple in your `settings.py` file. Did it ask you to create a superuser? Use "admin1234" for both your username and password, and 'admin@bloggy.com' for the email. If not, run `python manage.py createsuperuser` to create one.

You should now see a the `bloggy.db` file in your project's root directory.

Sanity check

Launch the Django development server:

```
$ python manage.py runserver
```

Open <http://localhost:8000/> and you should see the familiar, light-blue "Welcome to Django" screen. Kill the server.

Setup an App

Start a new app

```
$ python manage.py startapp blog
```

Setup the model

Define the model for your application using the Django ORM rather than with raw SQL.

To do so, add the following code to *models.py*:

```
from django.db import models

class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=100)
    content = models.TextField()
```

This class, `Post()`, which inherits some of the basic properties from the standard Django `Model()` class, defines the database table as well as each field - `created_at`, `title`, and `content`, representing a single blog post.

NOTE: Much like SQLAlchemy and web2py's DAL, the Django ORM provides a database abstraction layer used for interacting with a database via [Python objects](#).

Note that the primary key - which is a unique id (uuid) that we don't even need to define - and the `created_at` timestamp will both be automatically generated for us when new `Post` objects are added to the database. In other words, we just need to explicitly add the `title` and `content` when creating new objects (database rows).

settings.py

Add the new app to the *settings.py* file:

```
INSTALLED_APPS = (
    # Django Apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Local Apps
    'blog',
)
```

Migrations

Execute the underlying SQL statements to create the database tables by creating then applying the [migration](#):

```
$ python manage.py makemigrations blog
```

You should see something similar to:

```
Migrations for 'blog':
  0001_initial.py:
    - Create model Post
```

The `makemigrations` command essentially tells Django that changes have been made to the models and we want to create a migration. You can see the actual migration by opening the file `0001_initial.py` from the "migrations folder".

Now we need to apply the migration to create the *actual* database tables:

```
$ python manage.py migrate
```

You should see something like:

```
Operations to perform:
  Apply all migrations: admin, blog, contenttypes, auth, sessions
Running migrations:
  Applying blog.0001_initial... OK
```

Just remember whenever you want to make any changes to the database, you must:

1. Update your models
2. Create your migration - `python manage.py makemigrations`
3. Apply those migrations - `python manage.py migrate`

We'll discuss this workflow in greater detail further in this chapter.

Django Shell

Django provides an interactive Python shell for accessing the Django API. Use the following command to start the Shell:

```
$ python manage.py shell
```

Working with the database

Searching

Let's [add](#) some data to the table we just created via the database API. Start by importing the `Post` model we created in the Django Shell:

```
>>> python manage.py shell
>>> from blog.models import Post
```

If you search for objects in the table (somewhat equivalent to the SQL statement `select * from blog_post`) you should find that it's empty:

```
>>> Post.objects.all()
<QuerySet []>
```

Adding data

To add new rows, we can use the following commands:

```
>>> p = Post(title="What Am I Good At", content="What am I good at? What is my talent? What makes me stand out? These are the questions we ask ourselves over and over again and somehow can not seem to come up with the perfect answer. This is because we are blinded, we are blinded by our own bias on who we are and what we should be. But discovering the answers to these questions is crucial in branding yourself. You need to know what your strengths are in order to build upon them and make them better")
>>> p.save()
>>> p = Post(title="Charting Best Practices", content="Charting data and determining business progress is an important part of measuring success. From recording financial statistics to webpage visitor tracking, finding the best practices for charting your data is vastly important for your company's success. Here is a look at five charting best practices for optimal data visualization and analysis.")
>>> p.save()
>>> p = Post(title="Understand Your Support System Better With Sentiment Analysis", content="There's more to evaluating success than monitoring your bottom line. While analyzing your support system on a macro level helps to ensure your costs are going down and earnings are rising, taking a micro approach to your business gives you a thorough appreciation of your business' performance. Sentiment analysis helps you to clearly see whether your business practices are leading to higher customer satisfaction, or if you're on the verge of running clients away.")
>>> p.save()
```

NOTE: Remember that we don't need to add a primary key or the `created_at` time stamp as those are auto-generated.

Searching (again)

Now if you search for all objects, three objects should be returned:

```
>>> Post.objects.all()
<QuerySet [<Post: Post object>, <Post: Post object>, <Post: Post object>]>
```

NOTE: When we use the `all()` or `filter()` functions, a `QuerySet` (list) is returned, which is an [iterable](#).

Notice how `<Post: Post object>` returns absolutely no distinguishing information about the object. Let's change that.

Customizing the models

Open up your `models.py` file and add the following code:

```
def __str__(self):  
    return self.title
```

Your file should now look like this:

```
from django.db import models  
  
class Post(models.Model):  
    created_at = models.DateTimeField(auto_now_add=True)  
    title = models.CharField(max_length=100)  
    content = models.TextField()  
  
    def __str__(self):  
        return self.title
```

Save your `models.py` file, exit the Shell, re-open the Shell, import the `Post` model class again (`from blog.models import Post`), and now run the query `Post.objects.all()` .

You should now see:

```
<QuerySet [<Post: What Am I Good At>, <Post: Understand Your Support System Better  
With Sentiment Analysis>, <Post: Charting Best Practices>]>
```

This should be much easier to read and understand. We know there are three rows in the table, and we know their titles.

Want more?

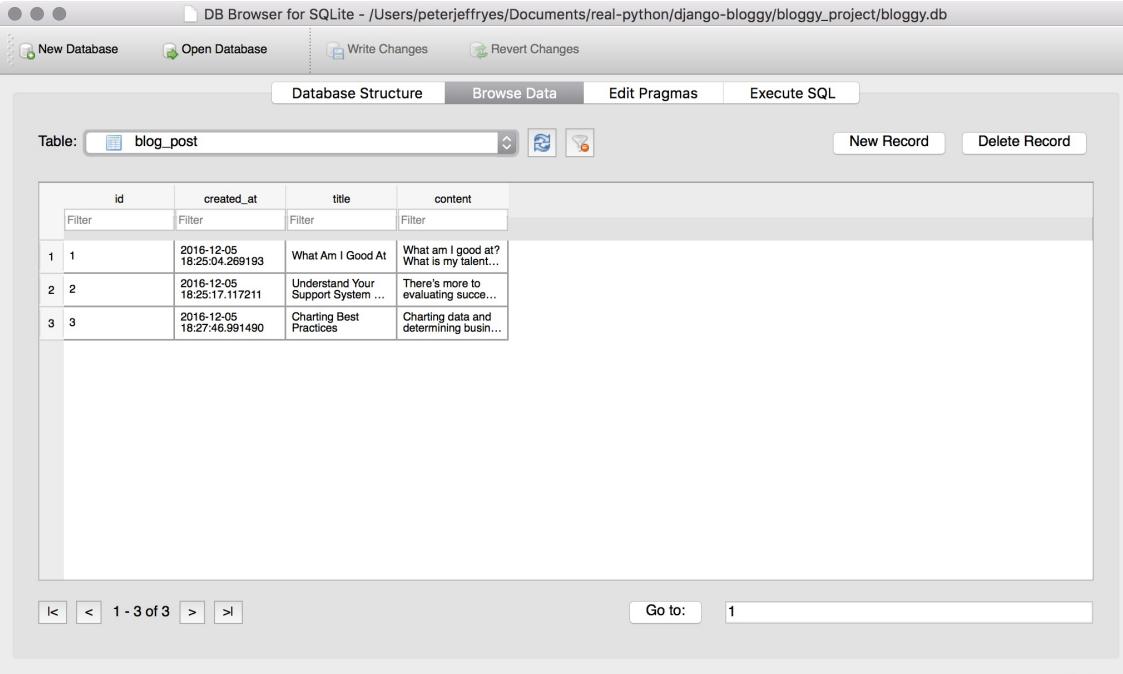
Depending on how much information you want returned, you could add all the fields to the `models.py` file:

```
def __str__(self):  
    return "{0}/{1}/{2}/{3}\n".format(self.id, self.created_at, self.title, self.c  
ontent)
```

Test this out. What does this return? Make sure to update this when you're done so it's just returning the title again:

```
def __str__(self):  
    return self.title
```

Open up your [SQLite Browser](#) to make sure the data was added correctly:



The screenshot shows the DB Browser for SQLite interface. The title bar reads "DB Browser for SQLite - /Users/peterjeffryes/Documents/real-python/django-bloggy/bloggy_project/bloggy.db". The main window displays a table named "blog_post" with four columns: "id", "created_at", "title", and "content". The data is as follows:

	id	created_at	title	content
1	1	2016-12-05 18:25:04.269193	What Am I Good At	What am I good at? What is my talent...
2	2	2016-12-05 18:25:17.117211	Understand Your Support System ...	There's more to evaluating succe...
3	3	2016-12-05 18:27:46.991490	Charting Best Practices	Charting data and determining busin...

At the bottom, there are navigation buttons (back, forward, search, etc.) and a "Go to:" input field with the value "1". The status bar at the bottom right shows "UTF-8".

Searching (slightly more advanced)

Let's look at how to query then filter the data from the Shell.

```
>>> from blog.models import Post
```

Return objects by a specific id:

```
>>> Post.objects.filter(id=1)
```

Return objects by ids > 1:

```
>>> Post.objects.filter(id__gt=1)
```

Return objects where the title contains a specific word:

```
>>> Post.objects.filter(title__contains='Charting')
```

If you want more info on querying databases via the Django ORM in the Shell, take a look at the official Django [documentation](#). And if you want a challenge, add more data from within the Shell via SQL and practice querying using straight SQL. Then delete all the data. Finally, see if you can add the same data and perform the same queries again within the Shell - but with objects via the Django ORM rather than SQL.

Homework

- Please read about the [Django Models](#) for more information on the Django ORM syntax.

Unit Tests for Models

Now that the database table is setup, let's write a quick unit test. It's common practice to write tests as you develop new models and views. Make sure to include at least one test for each function within your *models.py* files.

Add the following code to *tests.py*:

```
from django.test import TestCase
from blog.models import Post

class PostTests(TestCase):

    def test_str(self):
        my_title = Post(title='This is a basic title for a basic test case')
        self.assertEqual(
            str(my_title), 'This is a basic title for a basic test case',
        )
```

Run the test case:

```
$ python manage.py test blog -v 2
```

NOTE: You can run all the tests in the Django project with the following command
- `python manage.py test` ; or you can run the tests from a specific app, like in the command above.

You should see the following output:

```
reating test database for alias 'default' (':memory:')...
Operations to perform:
  Synchronize unmigrated apps: staticfiles, messages
  Apply all migrations: admin, blog, contenttypes, auth, sessions
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
  Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying blog.0001_initial... OK
  Applying sessions.0001_initial... OK
test_str (blog.tests.PostTests) ... ok

-----
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default' (':memory:')...
```

One thing to note is that since this test needed to add data to a database to run, it created a temporary, in-memory database and then destroyed it after the test ran. This prevents the test from accessing the real database and possibly damaging the database by mixing test data with real data.

NOTE: Anytime you make changes to an existing model or function, re-run the tests. If they fail, find out why. You may need to re-write them depending on the changes made. Or: the code you wrote may have broken the tests, which will need to be refactored.

Django Admin

Depending how familiar you are with the Django ORM and SQL in general, it's probably much easier to access and modify the data stored within the models using the [Django web-based admin](#). This is one of the most powerful built-in features that Django has to offer.

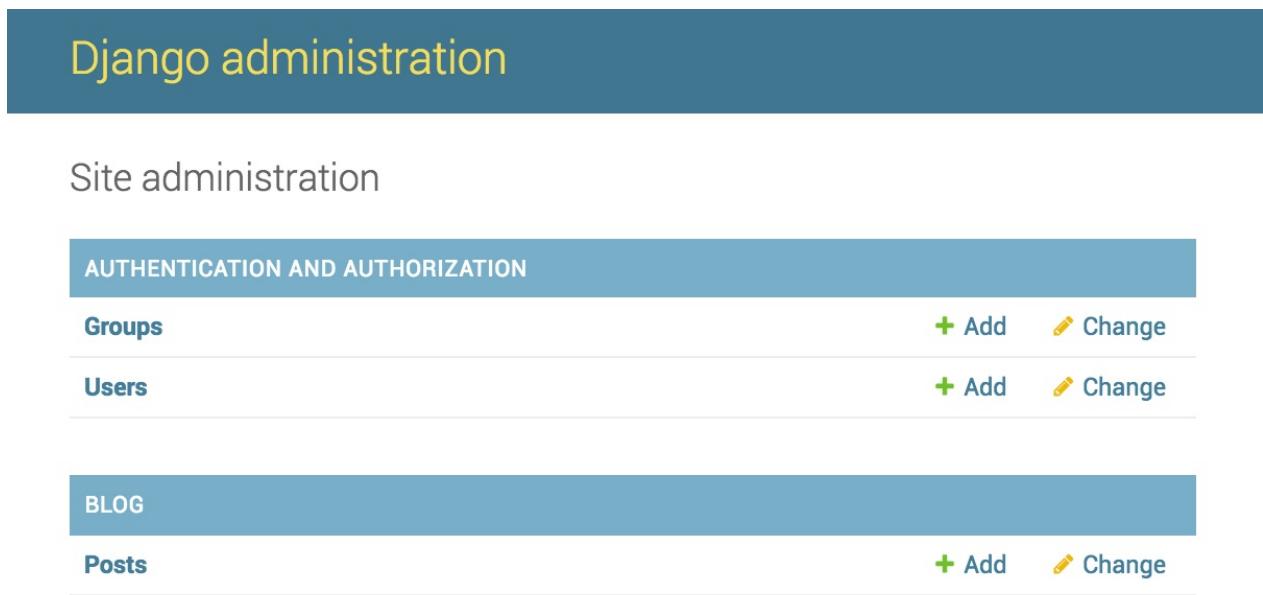
To access the admin, add the following code to the `admin.py` file in the "blog" directory:

```
from django.contrib import admin
from blog.models import Post

admin.site.register(Post)
```

Here, we're simply telling Django which models we want to make available to the admin.

Now let's access the Django Admin. Fire up the server and navigate to <http://localhost:8000/admin> within your browser. Enter your login credentials ("admin1234" and "admin1234").



The screenshot shows the Django Admin interface. At the top, a blue header bar contains the text "Django administration". Below it, a link "Site administration" is visible. The main content area has a light gray background. It features a blue header "AUTHENTICATION AND AUTHORIZATION" with two entries: "Groups" and "Users", each with a green "Add" button and a yellow "Change" button. Below this is another blue header "BLOG" with a single entry "Posts", also with a green "Add" button and a yellow "Change" button.

Add some more posts, change some posts, delete some posts. Go crazy.

Custom Admin View

We can customize the admin views by simply editing `admin.py`. For example, let's say that when we view the posts in the admin - <http://localhost:8000/admin/blog/post/>, we want to not only see the title, but the time the post was created at (from our `created_at` field in the model) as well.

Django makes this easy.

Start by creating a `PostAdmin()` class that inherits from `admin.ModelAdmin` from the `admin.py` within the "blog" directory:

```
# blog/admin.py

class PostAdmin(admin.ModelAdmin):
    pass
```

Next, in that new class, add a variable called `list_display`, and set it equal to a tuple that includes the fields from the database we want displayed:

```
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'created_at')
```

Finally, register this class with with Django's admin interface:

```
admin.site.register(Post, PostAdmin)
```

Updated `admin.py` code:

```
from django.contrib import admin
from blog.models import Post

class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'created_at')

admin.site.register(Post, PostAdmin)
```

View the changes at <http://localhost:8000/admin/blog/post/>.

Templates and Views

As mentioned, Django's views are equivalent to the controllers in most other MVC-style web frameworks. The views are generally paired with templates to generate HTML in the browser. Let's look at how to add the basic templates and views to our blog.

Before we start, create a new directory in the root called "templates" then within that directory add a "blog" directory, and then add the correct path to our `settings.py` file, just like we did with the 'Hello World' app:

```
'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

Your Project structure should now look like this:

```
└── bloggy_project
    ├── blog
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── migrations
    │   ├── models.py
    │   ├── tests.py
    │   └── views.py
    ├── bloggy.db
    ├── bloggy_project
    │   ├── __init__.py
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    ├── manage.py
    └── templates
        └── blog
```

View

Add the following code to the `views.py` file:

```
from django.http import HttpResponseRedirect
from django.template import Context, loader

from blog.models import Post

def index(request):
    latest_posts = Post.objects.all().order_by('-created_at')
    t = loader.get_template('blog/index.html')
    c = Context({'latest_posts': latest_posts, })
    return HttpResponseRedirect(t.render(c))
```

Template

Create a new file called *index.html* within the "templates/blog" directory and add the following code:

```
<h1>Bloggy: a blog app</h1>
{% if latest_posts %}
<ul>
{% for post in latest_posts %}
    <li>{{post.created_at}} | {{post.title}} | {{post.content}}</li>
{% endfor %}
</ul>
{% endif %}
```

URL

Add a *urls.py* file to your "bloggy_project/blog" directory, then add the following code:

```
from django.conf.urls import url
from blog import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

Now update the Project's *urls.py* file:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^blog/', include('blog.urls')),
]
```

Sanity Check

Test it out. Fire up the server and navigate to <http://localhost:8000/blog/>. You should have something that looks like this:

Bloggy: a blog app

- April 6, 2014, 7:32 p.m. | Understand Your Support System Better With Sentiment Analysis | There's more to evaluating success than monitoring your bottom line. While analyzing your support system on a macro level helps to ensure your costs are going down and earnings are rising, taking a micro approach to your business gives you a thorough appreciation of your business' performance. Sentiment analysis helps you to clearly see whether your business practices are leading to higher customer satisfaction, or if you're on the verge of running clients away.
- April 6, 2014, 7:31 p.m. | Charting Best Practices | Charting data and determining business progress is an important part of measuring success. From recording financial statistics to webpage visitor tracking, finding the best practices for charting your data is vastly important for your company's success. Here is a look at five charting best practices for optimal data visualization and analysis.
- April 6, 2014, 7:31 p.m. | What Am I Good At | What am I good at? What is my talent? What makes me stand out? These are the questions we ask ourselves over and over again and somehow can not seem to come up with the perfect answer. This is because we are blinded, we are blinded by our own bias on who we are and what we should be. But discovering the answers to these questions is crucial in branding yourself. You need to know what your strengths are in order to build upon them and make them better

Refactor

Let's clean this up a bit.

Template (second iteration)

Update *index.html*:

```
<html>
<head>
    <title>Bloggy: a blog app</title>
</head>
<body>
    <h2>Welcome to Bloggy!</h2>
    {% if latest_posts %}
        <ul>
            {% for post in latest_posts %}
                <h3><a href="/blog/{{ post.id }}">{{ post.title }}</a></h3>
                <p><em>{{ post.created_at }}</em></p>
                <p>{{ post.content }}</p>
                <br>
            {% endfor %}
        </ul>
    {% endif %}
</body>
</html>
```

If you refresh the page now, you'll see that it's easier to read. Also, each post title is now a link. Try clicking on the link. You should see a 404 error because we have not set up the URL route or the template. Let's do that now.

Notice how we used two kinds of curly braces within the template. The first, `{% ... %}`, is used for adding Python logic/expressions such as a `if` statements or `for` loops, and the second, `{{ ... }}`, is used for inserting variables or the results of an expression.

Finally, Before moving on, update the timezone to reflect where you live. Flip back to the *Django: Quick Start* chapter for more information.

post.html

URL

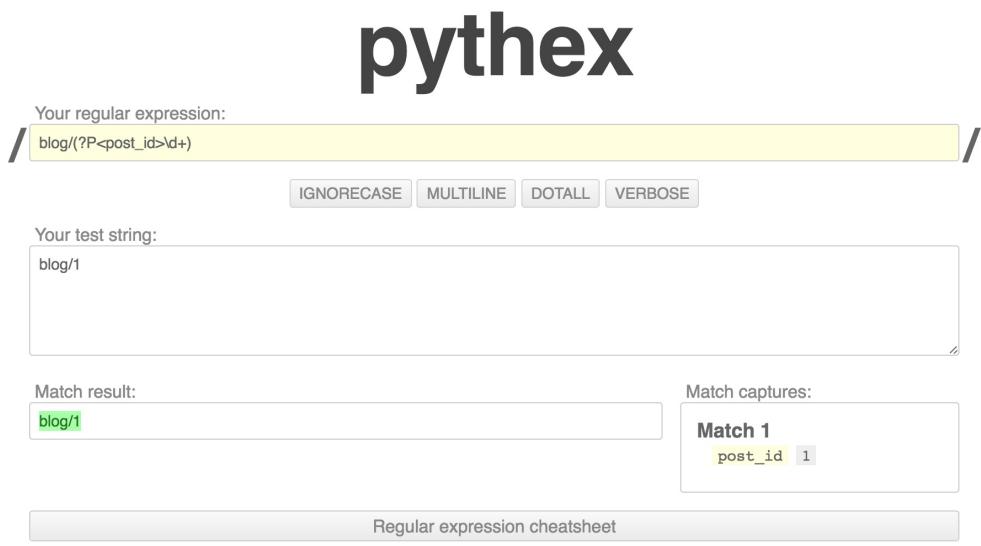
Update the `urls.py` file within the "blog" directory:

```
from django.conf.urls import url
from blog import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^(?P<post_id>\d+)/$', views.post, name='post'),
]
```

Take a look at the new URL. The pattern, `(?P<slug>\d+)`, is made up of regular expressions.

Turn to the the [Python docs](#) to learn more about regular expressions. There's also a chapter on regular expressions within the first Real Python course. Finally, you can test out regular expressions using [Pythex](#):



Inspired by [Rubular](#). For a complete reference, see the official [re module documentation](#).
 Made by [Gabriel Rodriguez](#). Powered by [Flask](#) and [jQuery](#).

View

Add a new function to `views.py`:

```
from django.shortcuts import get_object_or_404

def post(request, post_id):
    single_post = get_object_or_404(Post, pk=post_id)
    t = loader.get_template('blog/post.html')
    c = Context({'single_post': single_post, })
    return HttpResponseRedirect(t.render(c))
```

This function accepts two arguments:

1. The `request` object
2. The `post_id`, which is the number (primary key) parsed from the URL by the regular expression we setup

Meanwhile the `get_object_or_404()` method queries the database for objects by a specific type (`id`) and returns that object if found. If not found, it returns a 404 error code.

Template

Create a new template called `post.html`:

```
<html>
<head>
  <title>Bloggy: {{ single_post.title }}</title>
</head>
<body>
<h2>{{ single_post.title }}</h2>
<ul>
  <p><em>{{ single_post.created_at }}</em></p>
  <p>{{ single_post.content }}</p>
  <br/>
</ul>
<p>Had enough? Return <a href="/blog">home</a>.</p><br/>
</body>
</html>
```

Back on the development server, test out the links for each post. They should all be working now.

Friendly Views

Take a look at our current URLs for displaying posts - `/blog/1` , `/blog/2` , and so forth. Although this works, it's not the most human readable (or SEO friendly). Instead, let's update this so that the post title is used in the URL rather than the primary key.

To achieve this, we need to update our `views.py`, `urls.py`, and `index.html` files.

View

Update the `index()` function within the `views.py` file:

```
def index(request):
    latest_posts = Post.objects.all().order_by('-created_at')
    t = loader.get_template('blog/index.html')
    context_dict = {'latest_posts': latest_posts, }
    for post in latest_posts:
        post.url = post.title.replace(' ', '_')
    c = Context(context_dict)
    return HttpResponse(t.render(c))
```

The main difference is that we created a `post.url` using a `for` loop to replace the spaces in a post name with underscores:

```
for post in latest_posts:
    post.url = post.title.replace(' ', '_')
```

Thus, a post title of "test post" will convert to "test_post". This will make our URLs look better (and, hopefully, more search engine friendly). If we didn't remove the spaces or add the underscore (`post.url = post.title`), the URL would show up as "test%20post", which is difficult to read. Try this out if you're curious.

Template

Now update the actual URL in the `index.html` template...

Replace:

```
<h3><a href="/blog/{{ post.post_id }}">{{ post.title }}</a></h3>
```

With:

```
<h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
```

View (again)

The `post()` function is still searching for a post in the database based on an `id`. Let's update that:

```
def post(request, post_url):
    single_post = get_object_or_404(Post, title=post_url.replace('_', ' '))
    t = loader.get_template('blog/post.html')
    c = Context({'single_post': single_post, })
    return HttpResponse(t.render(c))
```

Now, this function is searching for the title in the database rather than the primary key. Notice how we have to replace the underscores with spaces so that it matches *exactly* what's in the database.

URL

Finally, let's update the regex in the `urls.py` file:

```
url(r'^(?P<post_url>\w+)/$', views.post, name='post'),
```

Here we changed the regular expression to match a sequence of alphanumeric characters before the trailing forward slash. Test this out with [Pythex](#).

Run the server. Now you should see URLs that are a little easier on the eye.

Django Migrations

Before moving on, let's look at how to handle database schema changes via [migrations](#) in Django.

NOTE: This feature came about from a KickStarter [campaign](#) just like all of the Real Python courses. Thank you all, again!

Before Django Migrations you had to use a package called [South](#), which wasn't always the easiest to work with. But now with migrations built-in, they automatically become part of your basic, everyday workflow.

Speaking of which, do you remember the workflow highlighted before:

1. Update your models in *models.py*
2. Create your migration - `python manage.py makemigrations`
3. Apply those migrations - `python manage.py migrate`

Let's go through it to make a change to our database.

Update your models

Open up your *models.py* file and add three new fields to the table:

```
tag = models.CharField(max_length=20, blank=True, null=True)
image = models.ImageField(upload_to="images", blank=True, null=True)
views = models.IntegerField(default=0)
```

NOTE: By setting `blank=True`, we are indicating that the field is not required and can be left blank within the form (or whenever data is inputted by the user). Meanwhile, `null=True` allows blank values to be stored in the database as `NULL`. These options are usually used in [tandem](#).

Since we're going to be working with images, we need to use [Pillow](#). You should have an error in your terminal telling you to do just that (if the development server is running):

```
django.core.management.base.SystemCheckError: SystemCheckError: System check identified some issues:

ERRORS:
blog.Post.image: (fields.E210) Cannot use ImageField because Pillow is not installed.

    HINT: Get Pillow at https://pypi.python.org/pypi/Pillow or run command "pip install Pillow".

System check identified 1 issue (0 silenced).
```

Install Pillow:

```
$ pip install Pillow==3.4.2
$ pip freeze > requirements.txt
```

Fire up your development server and log in to the admin page, and then try to add a new row to the `Post` model. You should see the error `table blog_post has no column named tag` because the fields we tried to add didn't get added to the database. That's a problem - and that's exactly where migrations come into play.

Create your migration

To create your migration simply run:

```
$ python manage.py makemigrations
Migrations for 'blog':
  0002_auto_20140918_2218.py:
    - Add field image to post
    - Add field tag to post
    - Add field views to post
```

Do you remember what the `makemigrations` command does? *It essentially tells Django that changes were made to the models and we want to create a migration.*

Adding on to that definition, Django scans your models and compares them to the newly created migration file, which you'll find in your "migrations" folder in the "blog" folder; it should start with `0002_`.

WARNING: You should always double-check the migration file to ensure that it is correct. Complex changes to the model may not always turn out the way you expect within the migration file. You can edit migration files directly, if necessary.

Apply migrations

Now, let's apply those migrations to the database:

```
$ python manage.py migrate
```

Output:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying blog.0002_auto_20161207_1607... OK
```

Did this work? Let's find out.

Run the server, enter the Django admin, and you now should be able to add new rows that include tags, an image, and/or the number of views.

Add a few posts with the new fields. *Don't add any images just yet.* Then update the remaining posts with tags and views. Do you remember how to update which fields are displayed in the admin? Add the number of views to the admin by updating the `list_display` tuple in the `admin.py` file:

```
list_display = ('title', 'created_at', 'views')
```

Update app

Now let's update the the application so that tags and images are displayed on the post page.

Update `post.html`:

```
<html>
<head>
  <title>Bloggy: {{ single_post.title }}</title>
</head>
<body>
<h2>{{ single_post.title }}</h2>
<ul>
  <p><em>{{ single_post.created_at }}</em></p>
  <p>{{ single_post.content }}</p>
  <p>Tag: {{ single_post.tag }}</p>
  <br>
  <p>
    {% if single_post.image %}
      
    {% endif %}
  </p>
</ul>
<p>Had enough? Return <a href="/blog">home</a>.</p><br/>
</body>
</html>
```

Update *settings.py* at the bottom of the file:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
STATIC_URL = '/static/'
```

Make sure to create these directories as well in the root.

Update Project-level *urls.py*:

```
from django.conf.urls import include, url
from django.contrib import admin
from .settings import MEDIA_ROOT
from django.views.static import serve

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^blog/', include('blog.urls')),
    url(r'^media/(?P<path>.*$', serve, {'document_root': MEDIA_ROOT}),
]
```

Add some more posts from the admin. Make sure to include images. Check out the results at <http://localhost:8000/blog/>. Also, you should see the images in the "media/images" folder.

View Counts

What happens to the view count when you visit a post? Nothing. It *should* increment, right?

Let's update that in the views:

```
def post(request, post_url):
    single_post = get_object_or_404(Post, title=post_url.replace('_', ' '))
    single_post.views += 1 # increment the number of views
    single_post.save()      # and save it
    t = loader.get_template('blog/post.html')
    c = Context({'single_post': single_post, })
    return HttpResponse(t.render(c))
```

Super simple.

Let's also add a counter to the post page by adding the following code to *post.html*:

```
<p>Views: {{ single_post.views }}</p>
```

Save. Navigate to a post and check out the counter. Refresh the page to watch it increment.

Styles

Before adding any styles, we need to break our templates into base and child templates, so that the child templates inherit the HTML and styles from the base template. We've covered this a number of times before so we won't go into great detail. If you would like more info as to how this is achieved specifically in Django, please see [this](#) document.

Parent template

Create a new template file called `_base.html` in the `templates` directory. This is the parent template. Get the code from the `assets` directory in the book 2 exercises [repo](#).

NOTE: Make sure to put the `_base.html` template in the main "templates" directory, while the other templates go in the "blog" directory. By doing so, you can now use this same base template for all your apps.

Child templates

Update `index.html`:

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

{% block content %}
{% if latest_posts %}
<ul>
  {% for post in latest_posts %}
    <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
    <p><em>{{ post.created_at }}</em></p>
    <p>{{ post.content }}</p>
    <br>
  {% endfor %}
</ul>
{% endif %}
{% endblock %}
```

Update `post.html`:

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

{% block content %}
<h2>{{ single_post.title }}</h2>
<ul>
<p><em>{{ single_post.created_at }}</em></p>
<p>{{ single_post.content }}</p>
<p>Tag: {{ single_post.tag }}</p>
<p>Views: {{ single_post.views }}</p>
<br>
<p>
{% if single_post.image %}
    
{% endif %}
</p>
</ul>
<p>Had enough? Return <a href="/blog">home</a>.</p><br/>
{% endblock %}
```

Take a look at the results. Amazing what five minutes - and [Bootstrap](#) - can do.

Popular Posts

Next, let's update the index view to return the top five posts by popularity (most views). If, however, there are five posts or less, all posts will be returned.

View

Update `views.py`:

```
def index(request):
    latest_posts = Post.objects.all().order_by('-created_at')
    popular_posts = Post.objects.order_by('-views')[:5]
    t = loader.get_template('blog/index.html')
    context_dict = {
        'latest_posts': latest_posts,
        'popular_posts': popular_posts,
    }
    for post in latest_posts:
        post.url = post.title.replace(' ', '_')
    c = Context(context_dict)
    return HttpResponse(t.render(c))
```

Template

Update the `index.html` template, so that it loops through the `popular_posts`:

```
<h4>Must See:</h4>
<ul>
    {% for popular_post in popular_posts %}
        <li><a href="/blog/{{ popular_post.title }}">{{ popular_post.title }}</a></li>
    {% endfor %}
</ul>
```

Updated file:

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

{% block content %}
{% if latest_posts %}
<ul>
  {% for post in latest_posts %}
    <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
    <p><em>{{ post.created_at }}</em></p>
    <p>{{ post.content }}</p>
    <br>
  {% endfor %}
</ul>
{% endif %}
<h4>Must See:</h4>
<ul>
  {% for popular_post in popular_posts %}
    <li><a href="/blog/{{ popular_post.title }}">{{ popular_post.title }}</a></li>
  {% endfor %}
</ul>
{% endblock %}
```

Bootstrap

Next let's add the [Bootstrap Grid System](#) to our *index.html* template:

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

{% block content %}


{% if latest_posts %}



{% for post in latest_posts %}
    

### {{ post.title }}



{{ post.created_at }}



{{ post.content }}

{% endfor %}


{% endif %}



#### Must See:




{% for popular_post in popular_posts %}
    - {{ popular\_post.title }}


```

SEE ALSO: For more info on how the Grid System works, please review the [Getting Started With Bootstrap 3](#) post.

View

If you look at the actual URLs for the popular posts, you'll see that they do not have the underscores in the URLs. Let's change that. The easiest way to correct this is to just add another `for` loop to the index view:

```
for popular_post in popular_posts:
    popular_post.url = popular_post.title.replace(' ', '_')
```

Updated function:

```
def index(request):
    latest_posts = Post.objects.all().order_by('-created_at')
    popular_posts = Post.objects.order_by('-views')[:5]
    t = loader.get_template('blog/index.html')
    context_dict = {
        'latest_posts': latest_posts,
        'popular_posts': popular_posts,
    }
    for post in latest_posts:
        post.url = post.title.replace(' ', '_')
    for popular_post in popular_posts:
        popular_post.url = popular_post.title.replace(' ', '_')
    c = Context(context_dict)
    return HttpResponse(t.render(c))
```

Then update the URL within the *index.html* file:

```
<li><a href="/blog/{{ popular_post.url }}">{{ popular_post.title }}</a></li>
```

Test this out. It should work.

Finally, let's add a `encode_url()` helper function to *views.py* and refactor the `index()` function to clean up the code:

```
# helper function
def encode_url(url):
    return url.replace(' ', '_')

def index(request):
    latest_posts = Post.objects.all().order_by('-created_at')
    popular_posts = Post.objects.order_by('-views')[:5]
    t = loader.get_template('blog/index.html')
    context_dict = {
        'latest_posts': latest_posts, 'popular_posts': popular_posts,
    }
    for post in latest_posts:
        post.url = encode_url(post.title)
    for popular_post in popular_posts:
        popular_post.url = encode_url(popular_post.title)
    c = Context(context_dict)
    return HttpResponse(t.render(c))
```

Test this out in your development server to make sure nothing broke. Try running your test suite as well:

```
$ python manage.py test -v 2
```

Next chapter, we will create a form so we can add posts to our blog.

Homework

- Update *post.html* file to show the popular posts.
- Notice how we have some duplicate code in both the `index()` and `post()` functions. This [stinks](#). How can you write a helper function for this so that you're only writing that code once?

Django Bloggy: A blog app (part two)

Our blog app is looking good, but we still can't create new content via the app. In this chapter we are going to add the ability to create blog posts with in the app.

Forms

In this section, we'll look at how to add forms to allow end-users to add posts utilizing Django's built-in [form handling](#) features.

Such features allow one to:

1. Display an HTML form with automatically generated form widgets (like a text field or date picker)
2. Check submitted data against a set of validation rules
3. Redisplay a form in case of validation errors
4. Convert submitted form data to the relevant Python data types

In short, forms take the inputted user data, validate it, and then convert the data to Python objects.

Since we have a database-driven application, we will be taking advantage of a particular type of form called a [ModelForm](#). These forms map the user input to specific columns in the database.

To simplify the process of form creation, let's split the workflow into four steps:

1. Create a `forms.py` file to add fields to the form
2. Add the handling of the form logic to the view
3. Add or update a template to display the form
4. Add a `urlpattern` for the new view

Create a `forms.py` file to add fields to your form

Include the following code in `blog/forms.py`:

```
from django import forms
from blog.models import Post

class PostForm(forms.ModelForm):

    class Meta:
        model = Post
        fields = ['title', 'content', 'tag', 'image', 'views']
```

Here, we created a new `ModelForm` that's mapped to our model via the `Meta()` inner class `model = Post`. Notice how each of our form fields have an associated column in the database. This is required.

Add the handling of the form logic to the View

Next, update the view for handling the form logic - e.g., displaying the form, saving form data, alerting the user about validation errors, etc.

Add the following code to `blog/views.py`:

```
def add_post(request):
    if request.method == 'POST':
        form = PostForm(request.POST, request.FILES)
        if form.is_valid(): # is the form valid?
            form.save(commit=True) # yes? save to database
            return redirect(index)
        else:
            print(form.errors) # no? display errors to end user
    else:
        form = PostForm()
    return render(request, 'blog/add_post.html', {'form': form})
```

Also be sure to update the imports:

```
from django.http import HttpResponseRedirect
from django.template import Context, loader, RequestContext
from django.shortcuts import get_object_or_404, render, redirect

from blog.models import Post
from blog.forms import PostForm
```

What's going on here?

First, we determine if the request is a GET or POST. If the former, the form is displayed, `form = PostForm()` ; and, if the latter, we process the form data:

```
form = PostForm(request.POST, request.FILES)
if form.is_valid(): # is the form valid?
    form.save(commit=True) # yes? save to database
    return redirect(index)
else:
    print(form.errors) # no? display errors to end user
```

If it's a POST request, we first determine if the supplied data is valid or not.

Essentially, forms have two different types of validation that are triggered when `is_valid()` is called on a form - field and form validation:

1. *Field validation*, which happens at the form level, validates the user inputs against the arguments specified in the `ModelForm` - i.e., `max_length=100` , `required=False` , etc. Be sure to look over the official Django Documentation on [Widgets](#) to see the available fields and the parameters that each can take.
2. Once the fields are validated, the values are converted over to Python objects and then *form validation* occurs via the form's `clean` method. Read more about this method [here](#).

Validation ensures that Django does not add any data to the database from a submitted form that could potentially harm your database.

Again, each of these forms of validation happen implicitly as soon as the `is_valid()` method is called. You can, however, customize the process. Read more about the overall process from the links above or for a more detailed workflow, please see the Django form documentation [here](#).

After the data is validated, Django either saves the data to the database, `form.save(commit=True)` and redirects the user to the index page or outputs the errors to the end user.

Add or update a template to display the form

Moving on, let's create the template called `add_post.html` within the "templates/blog" directory:

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

{% block content %}
<h2>Add a Post</h2>
<form id="post_form" method="post" action="/blog/add_post/" enctype="multipart/form-data">
    {% csrf_token %}
    {% for hidden in form.hidden_fields %}
        {{ hidden }}
    {% endfor %}

    {% for field in form.visible_fields %}
        {{ field.errors }}
        {{ label_tag }}
        {{ field }}
    {% endfor %}

    <input type="submit" name="submit" value="Create Post">
</form>
{% endblock %}
```

Here, we have a `<form>` tag, which loops through both the visible and hidden fields. Only the visible fields will produce markup (HTML). Read more about such loops [here](#).

Also, within the visible fields loop, we are displaying the validation errors, `{{ field.errors }}`, as well as the name of the field, `label_tag`. You never want to do this for the hidden fields since this will *really* confuse the end user. You just want to have tests in place to ensure that the hidden fields generate valid data each and every time. We'll discuss this in a bit.

Finally, did you notice the `{% csrf_token %}` tag? This is a Cross-Site Request Forgery (CSRF) token, which is required by Django. Please read more about it from the official Django [documentation](#).

Alternatively, if you want to keep it simple, you can render the form with one tag:

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

{% block content %}
<h2>Add a Post</h2>
<form id="post_form" method="post" action="/blog/add_post/" enctype="multipart/form-data">
    {% csrf_token %}

    {{ form }}

    <input type="submit" name="submit" value="Create Post">
</form>
{% endblock %}
```

When you test this out (which will happen in a bit), try both types of templates. Make sure to end with the latter one.

Since we're not adding any custom features to the form in the above (simple) template, we can get away with just this simplified template and get the same functionality as the more complex template. If you are interested in further customization, check out the [Django documentation](#).

NOTE: When you want users to upload files via a form, you must set the `enctype` to `multipart/form-data`. If you do not remember to do this, you won't get an error; the upload just won't be saved - so, it's vital that you remember to do this. You could even write your templates in the following manner to ensure that you include `enctype="multipart/form-data"` in case you don't know whether users will be uploading files ahead of time:

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post" action="">
{% else %}
    <form method="post" action="">
{% endif %}
{{ form }}
</form>
```

Add a `urlpattern` for the new view

Now we just need to map the view, `add_post`, to a URL:

```
from django.conf.urls import url
from blog import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^add_post/$', views.add_post, name='add_post'), # add post form
    url(r'^^(?P<slug>[\w\-\-]+)$', views.post, name='post'),
]
```

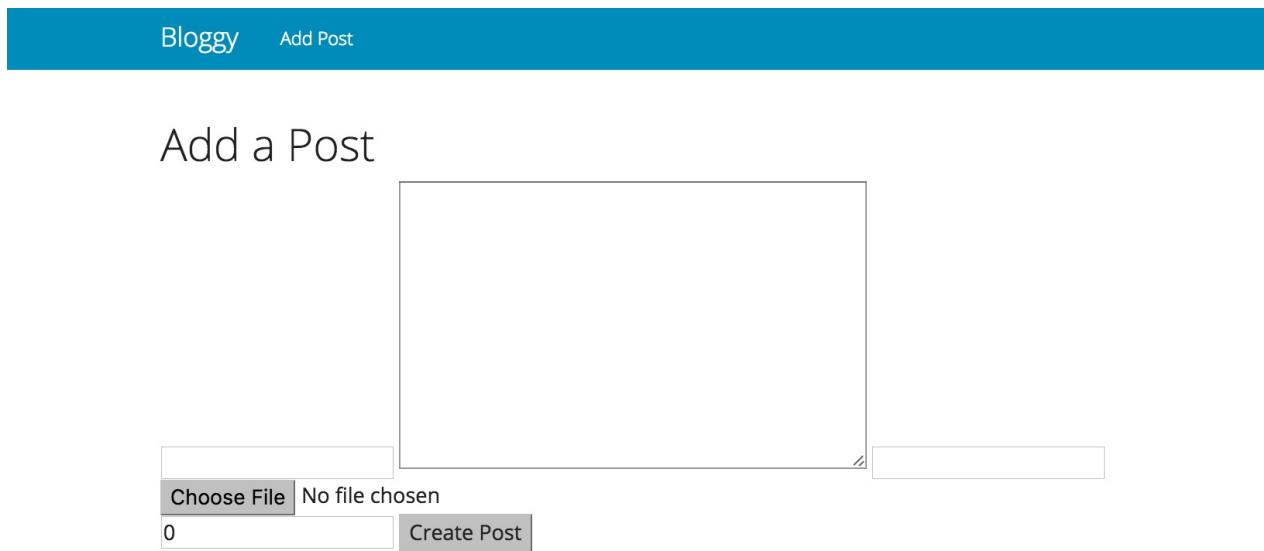
That's it. Just update the `Add Post` link in our `_base.html` template:

```
<li><a href="/blog/add_post/">Add Post</a></li>
```

Then test it out.

Styling Forms

Navigate to the form template at http://localhost:8000/blog/add_post/.



Looks pretty bad. We can clean this up quickly with Bootstrap, specifically with the [Django Forms Bootstrap](#) package.

Install:

```
$ pip install django-forms-bootstrap==3.0.1
$ pip freeze > requirements.txt
```

Add the package to your `INSTALLED_APPS` tuple within `settings.py`:

```
INSTALLED_APPS = (
    # Django Apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Third Party Apps
    'django_forms_bootstrap',

    # Local Apps
    'blog',
)
```

Update the template:

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

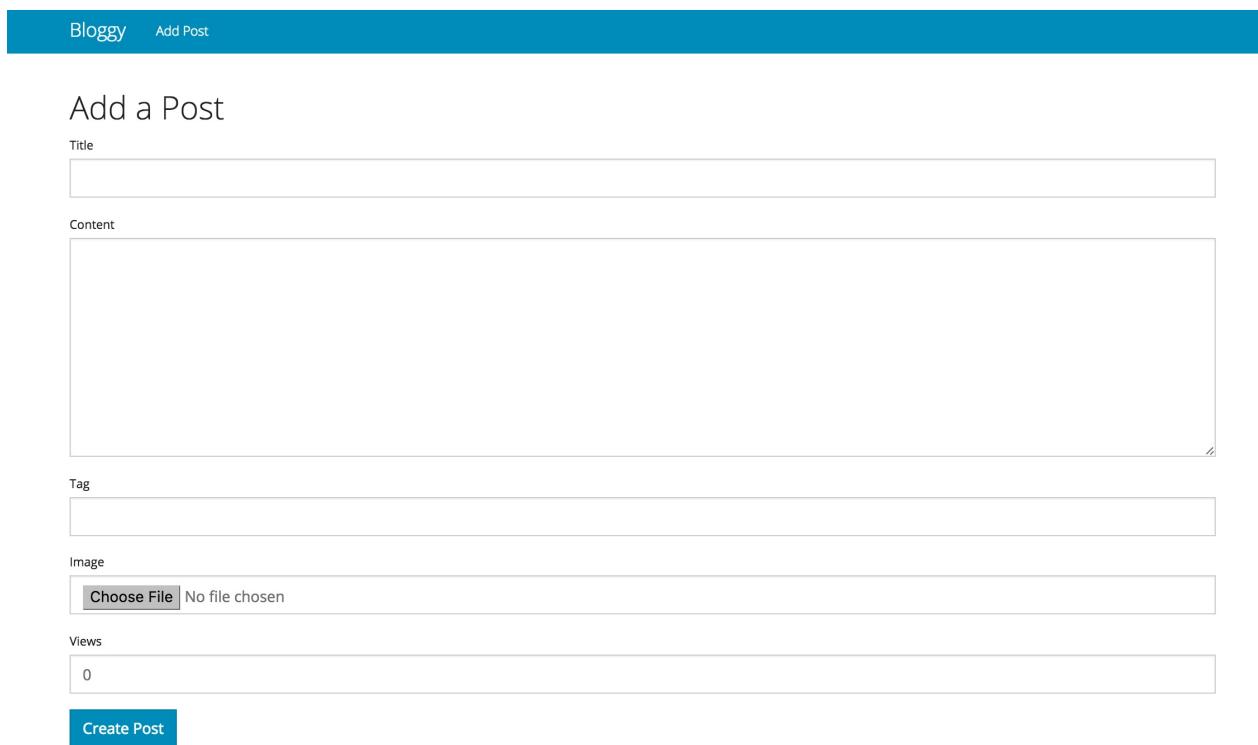
{% block content %}
{% load bootstrap_tags %}
<h2>Add a Post</h2>
<form id="post_form" method="post" action="/blog/add_post/" enctype="multipart/form-data">
    {% csrf_token %}

    {{ form | as_bootstrap}}

    <input type="submit" name="submit" value="Create Post" class="btn btn-primary">
</form>
<br>
{% endblock %}
```

Sanity check

Restart the server. Check out the results:



The screenshot shows a 'Add a Post' form. At the top, there's a header with 'Bloggy' and 'Add Post'. The form itself has several fields: 'Title' (a text input), 'Content' (a large text area), 'Tag' (a text input), 'Image' (a file input with a 'Choose File' button and 'No file chosen' text), 'Views' (a text input with '0' inside), and a 'Create Post' button at the bottom.

Removing a form field

Did you notice that when you add a new post that you can change the view count. We probably don't want end users messing with that, right?

Well, just remove that specific field from the `fields` list in `forms.py`:

```
from django import forms
from blog.models import Post

class PostForm(forms.ModelForm):

    class Meta:
        model = Post
        fields = ['title', 'content', 'tag', 'image']
```

Test it out.

Homework

- See if you can change the redirect URL after a successful form submission, `return redirect(index)`, to redirect to the newly created post.

Even Friendlier Views

What happens if you add two posts with the same title? When you try to view one of them, you should see this error:

```
get() returned more than one Post -- it returned 2!
```

Not good. Let's fix that by using a unique [slug](#) for the URL with the [Django Uuslug](#) package.

Install the package

```
$ pip install django-uuslug==1.1.8
$ pip freeze > requirements.txt
```

Model

Update *models.py*:

```
from django.db import models
from uuslug import uuslug

class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=100)
    content = models.TextField()
    tag = models.CharField(max_length=20, blank=True, null=True)
    image = models.ImageField(upload_to="images", blank=True, null=True)
    views = models.IntegerField(default=0)
    slug = models.CharField(max_length=100, unique=True)

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs):
        self.slug = uuslug(self.title, instance=self, max_length=100)
        super(Post, self).save(*args, **kwargs)
```

Essentially, we're overriding how the `slug` is saved by setting up the custom `save()` method. For more on this, please check out the official [Django Uuslug documentation](#).

What's next?

```
$ python manage.py makemigrations
```

However, it's not that easy this time. When you run that command, you'll see:

```
You are trying to add a non-nullable field 'slug' to post without a default;
we can't do that (the database needs something to populate existing rows).
Please select a fix:
1) Provide a one-off default now (will be set on all existing rows)
2) Quit, and let me add a default in models.py
Select an option:
```

Basically, since the field is not allowed to be blank, the existing database columns need to have a value. Let's provide a default - meaning that all columns will be given the same value - and then manually fix them later on. After all, this is the exact issue that we're trying to fix: All post URLs must be unique.

```
Select an option: 1
Please enter the default value now, as valid Python
The datetime module is available, so you can do e.g. datetime.date.today()
>>> "test"
Migrations for 'blog':
  0003_post_slug.py:
    - Add field slug to post
```

Before we can migrate this, we have another problem. Take a look at the new field we're adding:

```
slug = models.CharField(max_length=100, unique=True)
```

The value must be unique. If we try to migrate now, we will get an error since we're adding a default value of "test". Here's how we get around this: Open the migrations file and remove the unique constraint so that it looks like this:

```
class Migration(migrations.Migration):  
  
    dependencies = [  
        ('blog', '0002_auto_20140918_2218'),  
    ]  
  
    operations = [  
        migrations.AddField(  
            model_name='post',  
            name='slug',  
            field=models.CharField(default='test', max_length=100),  
            preserve_default=False,  
        ),  
    ]
```

Make sure the name of the dependency, 0002_auto_20140918_2218 , matches the previous migration file name.

To maintain consistency, let's also remove the constraint from *models.py*:

```
slug = models.CharField(max_length=100)
```

Push the migration through:

```
$ python manage.py migrate  
Operations to perform:  
  Synchronize unmigrated apps: django_forms_bootstrap  
  Apply all migrations: admin, blog, contenttypes, auth, sessions  
Synchronizing apps without migrations:  
  Creating tables...  
  Installing custom SQL...  
  Installing indexes...  
Running migrations:  
  Applying blog.0003_post_slug... OK
```

Now, let's update those fields to make them unique.

Django Shell

Start the Shell:

```
$ python manage.py shell
```

Grab all the posts, then loop through them appending the post title to a list:

```
>>> from blog.models import Post
>>> all_posts = Post.objects.all()
>>> titles = []
>>> for post in all_posts:
...     titles.append(post.title)
...
>>> print(titles)
['test', 'Charting Best Practices', 'Understand Your Support System Better With Se
ntiment Analysis', 'testing', 'pic test', 'look at this', 'hey']
```

Now, we need to update the `slug` field like so:

```
>>> from uuslug import slugify
>>> x = 1
>>> y = 0
>>> len(titles)
8
>>> while x <= 8:
...     Post.objects.filter(id=x).update(slug=slugify(titles[y]))
...     x += 1
...     y += 1
...
1
1
1
1
1
1
1
1
>>>
```

Exit the shell.

Model

Update the `slug` field in the model:

```
slug = models.CharField(max_length=100, unique=True)
```

Create a new migration, then apply the changes.

URL

Update the following URL pattern in `urls.py`:

```
url(r'^(?P<post_url>\w+)/$', views.post, name='post'),
```

To:

```
url(r'^(?P<slug>[\w|\-]+)$', views.post, name='post'),
```

Here we matched the `slug` with any number of '-' characters followed by any number of alphanumeric characters. Again, if you're unfamiliar with regex, check out the chapter on regular expressions in the first Real Python course and be sure to test out regular expressions with [Pythex](#).

View

Now we can simplify the `views.py` file (making sure to replace `post_url` with `slug`). Head over to the assets folder in the exercises [repo](#) for the updated code.

Can you spot the big difference?

Template

Now update the actual URL in the `index.html` template...

Replace:

```
<h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
```

With:

```
<h3><a href="/blog/{{ post.slug }}">{{ post.title }}</a></h3>
```

Make sure to update the popular posts URL in both `index.html` and `post.html` as well...

Replace:

```
<li><a href="/blog/{{ popular_post.url }}">{{ popular_post.title }}</a></li>
```

With:

```
<li><a href="/blog/{{ popular_post.slug }}">{{ popular_post.title }}</a></li>
```

Sanity Check

What happens now when you try to register a post with a duplicate title, like - "test"? You should get two unique URLs:

1. <http://localhost:8000/blog/test/>
2. <http://localhost:8000/blog/test-1/>

Bonus! What happens if you register a post with a symbol in the title, like - '?', '*', or '!'? It should drop it from the URL altogether.

Nice.

Stretch Goals

What else could you add to this app? Comments? User auth? *Tests!* Add whatever you feel like. Find tutorials on the web for assistance. Show us the results.

Homework

- Optional: You're ready to go through the Django [tutorial](#). For beginners, this tutorial can be pretty confusing. However, since you now have plenty of web development experience and have already created a basic Django app, you should have no trouble. Have fun! Learn something.

Django Workflow

The following is a basic workflow that you can use as a quick reference for developing a Django 1.10 project.

NOTE: We followed this workflow during the Hello World app and loosely followed this for our Blog app. Keep in mind that this is just meant as a guide, so alter this workflow as you see fit for developing your own app.

Setup

1. Within a new directory, create and activate a new virtual environment.
2. Install Django.
3. Create your project: `django-admin.py startproject <name>`
4. Create a new app: `python manage.py startapp <appname>`
5. Add your app to the `INSTALLED_APPS` tuple.

Add Basic URLs and Views

1. Map your Project's `urls.py` file to the new app.
2. In your app directory, create a `urls.py` file to define your app's URLs.
3. Add views, associated with the URLs, in your app's `views.py`; make sure they return a `HttpResponse` object. Depending on the situation, you may also need to query the database to get the required data requested by the end user.

Templates and Static Files

1. Create a `templates` and `static` directory within your project root.
2. Update `settings.py` to include the paths to your templates.
3. Add a template (HTML file) to the `templates` directory. Within that file, you can include the static file with - `{% load static %}` and `{% static "filename" %}`. Also, you may need to pass in data requested by the user.
4. Update the `views.py` file as necessary.

Models and Databases

1. Update the database engine in `settings.py` (if necessary, as it defaults to SQLite).

2. Create and apply a new migration.
3. Create a super user.
4. Add an `admin.py` file in each app that you want access to in the Admin.
5. Create your models for each app.
6. Create and apply a new migration. (Do this whenever you make *any* change to a model).

Forms

1. Create a `forms.py` file in the app directory to define form-related classes; define your `ModelForm` classes there.
2. Add or update a view for handling the form logic - e.g., displaying the form, saving the form data, alerting the user about validation errors, etc.
3. Add or update a template to display the form.
4. Add a `urlpattern` in the app's `urls.py` file for the new view.

User Registration

1. Create a `UserForm`
2. Add a view for creating a new user.
3. Add a template to display the form.
4. Add a `urlpattern` for the new view.

User Login

1. Add a view for handling user credentials.
2. Create a template to display a login form.
3. Add a `urlpattern` for the new view.

Setup the Template Structure

1. Find the common parts of each page (i.e., header, sidebar, footer).
2. Add these parts to a base template
3. Create specific templates that inherit from the base template.

Bloggy Redux: Introducing Blongo

Let's build a blog. Again. This time with [MongoDB](#)!

MongoDB

[MongoDB](#) is an open-source, document-oriented database. Classified as a NoSQL database, Mongo stores semi-structured data in the form of binary JSON objects (BSON). It's essentially schema-less and meant to be used for hierarchical data that does not conform to the traditional relational databases. Typically, if the data you are trying to store does not easily fit on to a spreadsheet, you could consider using Mongo or some other NoSQL database. For more on Mongo, check out the official [Mongo documentation](#).

To download, visit the following [site](#). Follow the instructions for your particular operating system.

You can check your version by running the following command:

```
$ mongo --version
MongoDB shell version: 3.4.0
```

Once complete, be sure to follow the instructions to get `mongod` running, which, according to the [official documentation](#), *is the primary daemon process for the MongoDB system. It handles data requests, manages data format, and performs background management operations.*

This must be running in order for you to connect to the Mongo server.

Now, let's get our Django Project going.

Talking to Mongo

In order for Python to talk to Mongo, you need some sort of ODM (Object Document Module), which is akin to an ORM for relational databases. There's a number to choose from, but we'll be using [MongoEngine](#), which uses [PyMongo](#) to connect to MongoDb. We'll use PyMongo to write some test scripts to learn how Python and Mongo talk to each other.

Create a new directory called "django-blongo", enter the newly created directory, activate a virtual environment, and then install MongoEngine and PyMongo:

```
$ pip install mongoengine==0.9
$ pip install pymongo==2.8
$ pip freeze > requirements.txt
```

NOTE: It is important that you use the versions specified above. Some of the newer versions of mongoengine and pymongo are not compatible with each other or do not support django out of the box.

Now, let's test the install in the Python Shell:

```
$ python
>>> import pymongo
>>> client = pymongo.MongoClient("localhost", 27017)
>>> db = client.test
>>> db.name
'test'
>>> db.my_collection
Collection(Database(MongoClient(host=['localhost:27017']), document_class=dict, tz_aware=False, connect=True), 'test', 'my_collection')
>>> db.my_collection.save({"django": 10})
ObjectId('53e2e5a03386b7202c8bde61')
>>> db.my_collection.save({"flask": 20})
ObjectId('53e2e5b13386b7202c8bde62')
>>> db.my_collection.save({"web2py": 30})
ObjectId('53e2e5bc3386b7202c8bde63')
>>> db.my_collection
Collection(Database(MongoClient(host=['localhost:27017']), document_class=dict, tz_aware=False, connect=True), 'test', 'my_collection')
>>> db.my_collection.find()
<pymongo.cursor.Cursor object at 0x10124f810>
>>> for item in db.my_collection.find():
...     print(item)
...
{'django': 10, '_id': ObjectId('5848da7425dc9ab950b64086')}
{'flask': 20, '_id': ObjectId('5848da8a25dc9ab950b64087')}
{'web2py': 30, '_id': ObjectId('5848da9325dc9ab950b64088')}
```

Here, we created a new database called `test`, and then added a new collection (which is equivalent to a table in relational databases). Then within that collection we saved a number of objects, which we queried for at the end. Simple, right?

Play around with this some more.

Try creating a new database and collection and adding in JSON objects to define a simple blog with a 'title' and a 'body'. Add a number of *short* blog posts to the collection.

Test Script

```
import pymongo

# Open the MongoDB connection
conn = pymongo.MongoClient('mongodb://localhost:27017')

# Print the available MongoDB databases
databases = conn.database_names()
for database in databases:
    print(database)

# Close the MongoDB connection
conn.close()
```

Save this as *mongo.py*. This simply connects to your instance of Mongo, then outputs all database names. It will come in handy.

Django Setup

Install Django within "django-blongo", then create a new Project:

```
$ pip install django==1.10.4
$ django-admin.py startproject blongo_project
$ cd blongo_project/
```

Update *settings.py*:

```
# DATABASES = {
#     'default': {
#         'ENGINE': 'django.db.backends.sqlite3',
#         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
#     }
# }

AUTHENTICATION_BACKENDS = (
    'mongoengine.django.auth.MongoEngineBackend',
)

SESSION_ENGINE = 'mongoengine.django.sessions'

MONGO_DATABASE_NAME = 'blongo'

from mongoengine import connect
connect(MONGO_DATABASE_NAME)
```

NOTE: One thing to take note of here is that by using MongoDB, you cannot use the `syncdb` or `migrate` commands. Thus, you lose out-of-the-box migrations as well as the Django Admin panel. There are a number of drop-in replacements for the Admin panel, like [django-mongoadmin](#). There are also a number of example scripts floating around on Github and Stack Overflow, detailing how to migrate your data.

Setup an App

Start a new app:

```
$ python manage.py startapp blongo
```

Add the new app to the `settings.py` file:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blongo',
)
```

Define the model for the application. Despite the fact that Mongo does not require a schema, it's still import to define one at the application level so that we can specify datatypes, force require fields, and create utility methods. Keep in mind that this is only *enforced* at the application-level, not within MongoDB itself. For more information, please see the official MongoEngine [documentation](#). Simply add the following code to the `models.py` file:

```
from mongoengine import Document, StringField, DateTimeField
import datetime

class Post(Document):
    title = StringField(max_length=200, required=True)
    content = StringField(required=True)
    date_published = DateTimeField(default=datetime.datetime.now, required=True)
```

Here, we just created a class called `Posts()` that inherits from `Document`, then we added the appropriate `fields`. Compare this model, to the model from the *Bloggy* chapter. There's very few differences.

Add Data

Let's go ahead and add some data to the `Posts` collection from the Django Shell:

```
$ python manage.py shell
>>> from blongo.models import Post
>>> import datetime
>>> Post(title='just a test', content='again, just a test', date_published=datetime.datetime.now()).save()
<Post: Post object>
```

To access the data we can use the `objects` attribute from the `Document()` class:

```
>>> for post in Post.objects:
...     print post.title
...
just a test
```

Now that we've got data in MongoDB, let's finish building the app. You should be pretty familiar with the process by now; if you have questions that are not addressed, please refer to the *Bloggy* chapter.

Update Project

Project URLs

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^$', include('blongo.urls')),
]
```

App URLs

```
from django.conf.urls import url
from blongo import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

Views

```
from django.http import HttpResponseRedirect
from django.template import Context, loader

from blongo.models import Post

def index(request):
    latest_posts = Post.objects
    t = loader.get_template('index.html')
    context_dict = {'latest_posts': latest_posts}
    c = Context(context_dict)
    return HttpResponseRedirect(t.render(c))
```

As you saw before, we're using the `objects` attribute to access the posts, then passing the `latest_posts` variable to the templates.

Template Settings

Add the paths for the template directory in `settings.py`:

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
]
```

Then add the directory to the Project:

```
.
├── blongo
│   ├── __init__.py
│   ├── admin.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── blongo_project
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
└── templates
```

Template

Finally add the template (`index.html`). Go to the "assets" folder in the [Book 2 Exercises repo](#) for the code.

Test

Fire up the development server, and then navigate to <http://localhost:8000/> to see the blog.

Test Script

Finally, let's update the test script to tailor it specifically to our app:

```
import pymongo

def get_collection(conn):

    databases = conn.database_names()

    if 'blongo' in databases:
        # connect to the blongo database
        db = conn.blongo
        # grab all collections
        collections = db.collection_names()
        # output all collection
        print("Collections")
        print("-----")
        for collection in collections:
            print(collection)
        # pick a collection to view
        col = input(
            '\nInput a collection name to show its field names: '
        )
        if col in collections:
            get_items(db[col].find())
        else:
            print("Sorry. The '{}' collection does not exist.".format(col))
    else:
        print("Sorry. The 'blongo' database does not exist.")

    # Close the MongoDB connection
    conn.close()

def get_items(collection_name):

    for items in collection_name:
        print(items)

if __name__ == '__main__':
    # Open the MongoDB connection
    conn = pymongo.Connection('mongodb://localhost:27017')
    get_collection(conn)
```

Run this to connect to MongoDB, grab the 'blongo' database, and then output the items within a specific collection. It tests out the 'post' collection.

Conclusion

Next Steps

Implement *all* of the features of *Bloggy* on your own. Good luck! With Mongo now integrated, you can add in the various features in the exact same way you we did before; you will just need to tailor the actual DB queries to the MongoEngine syntax. You should be able to hack your way through this to get most of the features working. If you get stuck or want to check your answers, just turn back to the *Bloggy* chapter.

Once done, email us the link to your Github project and we'll look it over.

Summary

There are some problems with how we integrated MongoDB into our Django project.

1. We do not have a basic admin to manage the data. Despite the ease of adding, updating, and deleting data within a Mongo collection, it's still nice to be able to have a GUI to manage this. How can you fix this?
 - At the **Application layer**: As noted, there are a number of Admin replacements, like [django-mongoadmin](#), [django-mongonaut](#), [Django non-rel](#). You could also build your own basic admin based on the *mongo.py* script.
 - At the **Database layer**: There are a number of GUI utilities to view and update database and collection info. Check out [Robomongo](#)
2. Think about testing as well: If you try to use the base Django [TestCase](#) to run your tests, they will immediately fail since it won't find the database in *settings.py*.

Fortunately, we've solved this problem for you in the next course. Cheers!

Django: Ecommerce Site

Thus far you've learned the web fundamentals from the ground up, hacked your way through a number of different exercises, asked questions, tried new things, and built some cool web apps. Now it's time to put the things you've learned together and build a practical, real-world application, using the principles of rapid web development. After you complete this project, you should be able to take your new skills out into the real world and develop your own projects.

As you probably guessed, we will be developing a basic e-commerce site.

NOTE: The point of this chapter is to get a working application up quickly. This application is extended throughout Course 3 into an enterprise-level application.

First, let's talk about rapid web development.

Rapid Web Development

What exactly is rapid web development? Well, as the name suggests it's about building a web site or application quickly and efficiently. The focus is on efficiently. It's relatively easy to develop a site quickly, but it's another thing altogether to develop a site quickly that meets the functional needs and requirements that your potential users demand.

As you have seen, development is broken into logical chunks. If you were starting from complete scratch we'd begin with prototyping to define the major features and design a mock-up through iterations: Prototype, Review, Refine. Repeat. After each stage you generally get more and more detailed, from low to high-fidelity, until you've hashed out all the features and prepared a mock-up complex enough to add a web framework, in order to apply the MVC-style architecture.

If you are beginning with low-fidelity, start prototyping with a pencil and paper. Avoid the temptation to use prototyping software until the latter stages, as these can restrict creativity.

As you define each function and apply a design, put yourself in the end users' shoes. What can they see? What can they do? Do they really care? For example, if one of your app's features is to allow users to view a list of search results from an airline aggregator in pricing buckets, what does this look like? Are the results text or graphic-based? Can the user drill down on the ranges to see the prices at a more granular level? Will the end user care? They better. Will this differentiate your product versus the competition? Perhaps not. But there should be some features that do separate your product from your competitor's products. Or perhaps your functionality is the same - you just implement it better?

Finally, rapid web development is one of the most important skills a web developer can have, especially developers who work within a start-up environment. Speed is the main advantage that start-ups have over their larger, more established competition. The key is to understand each layer of the development process, from beginning to end.

SEE ALSO: For more on prototyping, check out [this](#) excellent resource.

Prototyping

Prototyping is the process of building a working model of your application, from a front-end perspective, allowing you to test and refine your application's main features and functions. Again, it's common practice to begin with a low-fidelity prototype.

From there you can start building a storyboard, which traces the users' movements. For example, if the user clicks the action button and they are taken to a new page, create that new page. Again, for each main feature, answer these three questions:

1. What can they see?
2. What can they do?
3. Do they really care?

If you're building out a full-featured web application, take the time to define in detail every interaction the user has with the app. Create a storyboard, and then after plenty of iterations, build a high-fidelity prototype. One of the quickest means of doing this is with [Bootstrap](#).

Get used to using user stories to describe behavior, as discussed in the Flask BDD chapter. Break down each of your app's unique pieces of functionality into user stories to drive out the app's requirements. This not only helps with organizing the work, but tracking progress as well.

Follow this form:

```
As a <role>
I want <goal>
In order to <benefit>
```

For example:

```
As a visitor
I want to sign up
In order to view blog posts
```

Each of these individual stories are then further broken down until you can transfer them over to individual functions. This helps answer the question, "Where should I begin developing?"

We won't be getting this granular in this chapter since you now have the skills to do this on your own. For now, let's assume we already went through the prototyping phase, built out user stories, and figured out our app's requirements. We have also put this basic app, with little functionality on the web somewhere and validated our business model. We know we have something, in other words; we just need to build it. This is where Django comes in.

NOTE: In many cases, development begins with defining the model and constructing the database first. Since we're creating a rapid prototype, we'll start with the front-end first, adding the most important functions, then move to the back-end. This is vital for when you develop your basic minimum viable prototype/product (MVP). The goal is to create an app quickly to validate your product and business model. Once validated, you can finish developing your product, adding all components you need to transform your prototype into a project.

Project Setup

Create a new directory called "django_ecommerce". Create and activate a virtual environment within that directory, and then install Django:

```
$ pip install django==1.10.4
```

Start a new Django project:

```
$ django-admin.py startproject django_ecommerce
```

Set up a new app:

```
$ cd django_ecommerce
$ python manage.py startapp main
```

NOTE: Due to the various features of this project, we will be creating a number of different apps. Each app will play a different role within your main project. This is a good practice: Each app should encompass a single piece of functionality.

Your project structure should now look like this:

```
└── django_ecommerce
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── main
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── manage.py
```

Add your app to the `INSTALLED_APPS` section of `settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'main'
]
```

Create a new directory within the root directory called "templates", and then add the absolute path to the `settings.py` file within the `TEMPLATES` list:

```
'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

Add this line in `urls.py` directly after the imports:

```
admin.autodiscover()
```

Don't forget to add your Django project folder to Sublime as a new project.

WARNING: Remember, your development environment should be isolated from the rest of your development machine (via virtual environment) so your dependencies don't clash with one another. In addition, you should recreate the production environment as best you can on your development machine - meaning, you should use the same dependency versions in production as you do in development. We will set up a `requirements.txt` file later to handle this. Finally, it may be necessary to develop on a virtual machine to recreate the exact OS and even the same OS release so that bugs found within production are easily reproducible in development. The bigger the app, the greater the need for this. Much of this will be addressed in the next course.

Landing Page

Now that the main app is up, let's quickly add a landing page. Add the following code to the *main* app's *views.py* file:

```
from django.shortcuts import render

def index(request):
    return render(
        request, 'index.html'
    )
```

`index()` takes a parameter, `request`, which is an object that has info about the user requesting the page from the browser. The function's response is to simply render the *index.html* template. In other words, when a user navigates to the *index.html* page (the request), the Django controller renders the *index.html* template (the response).

NOTE: Did you notice the `render()` **function**? This is simply used to render the given template.

Next, we need to add a new pattern to our Project's *urls.py*:

```
from main import views as main_views

url(r'^$', main_views.index, name='home'),
```

Finally, we need to create the *index.html* template. Create a new file called *base.html* (the parent template) within the templates directory. Go to the exercises [repo](#) and look in the assets for the code.

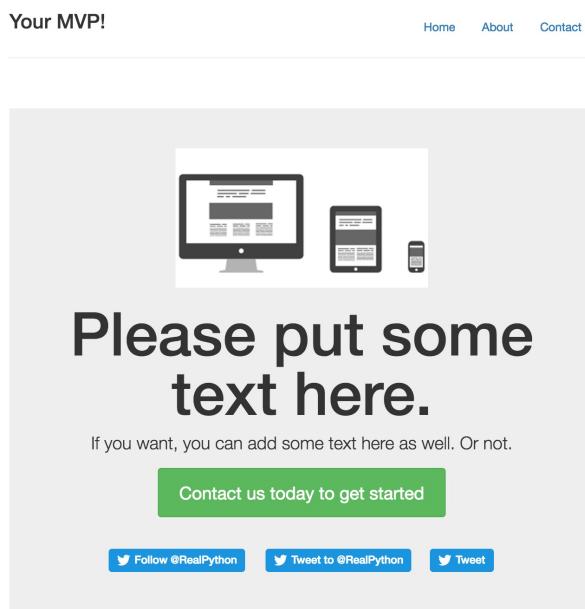
Do the same for the *index.html* (child) template.

By now you should understand the relationship between the parent and child templates so we won't go over that again. Also, your project structure should now look like this:

```
└── django_ecommerce
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── main
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── manage.py
└── templates
    ├── base.html
    └── index.html
```

Fire up your server:

```
$ python manage.py runserver
```



Looking good so far, but let's add some more pages. Before that, though, let's take a step back and talk about Bootstrap.

Bootstrap

We used the [Bootstrap](#) front end framework to quickly add a basic design. To avoid looking too much like, well, a generic Bootstrap site, the design should be customized, which is not too difficult to do. In fact, as long as you have a basic understanding of CSS and HTML, you shouldn't have a problem. That said, customizing Bootstrap can be time-consuming. It takes practice to get good at it. Try not to get discouraged as you build out your prototype. Work on one single area at a time, then take a break. Remember: It does not have to be perfect - it just has to give users, a better sense of how your application works.

Make as many changes as you want. Learning CSS like this is a trial and error process: You just have to make a few changes, then refresh the browser, see how they look, and then make more changes or update or revert old changes. Again, it takes time to know what will look good. Eventually, after much practice, you will find that you will be spending less and less time on each section, as you know how to get a good base set up quickly and then you can focus on creating something unique.

In this chapter, we will not focus too much time on the design process. Be sure to check out the next course where we had a slick Star Wars theme to our application.

SEE ALSO: If you're working on your own application, [Jetstrap](#) is a great resource for creating quick Bootstrap prototypes. Try it out. Also, check out [this](#) for info on how to create a nice sales page with Bootstrap.

About Page

First, let's add the [Flatpages App](#), which will allow us to add basic pages with HTML content. Add it to the `INSTALLED_APPS` section in `settings.py`:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.sites',  
    'django.contrib.flatpages',  
    'main'  
]
```

And also add in `settings.py` (where it goes is not important):

```
SITE_ID = 1
```

Update the `MIDDLEWARE_CLASSES` of `settings.py`:

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
    'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',  
]
```

Wait. What is middleware? Check the [Django docs](#) for a detailed answer specific to Django's middleware. Read it even though it may not make much sense right now. Then check the *Modern Web Development* chapter for a more general definition. Finally, once you complete this course, go through the third Real Python course, *Advanced Web Development with Django*, for a detailed explanation specific to Django, which will tie everything together.

Add the following pattern to `urls.py`:

```
url(r'^pages/', include('django.contrib.flatpages.urls')),
```

Update the import as well:

```
from django.conf.urls import url, include
```

Then, add a new template folder within the "templates" directory called "flatpages". Add a default template by creating a new file, *default.html*, within that new directory. Add the following code to the file:

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

{% block content %}
{{ flatpage.content }}
{% endblock %}
```

Sync the database and create a superuser:

```
$ python manage.py migrate
```

NOTE: If you were not prompted to create a superuser after your migrate, run this command: 'python manage.py createsuperuser'

Update the `About` url in the *base.html* file:

```
<li><a href="/pages/about">About</a></li>
```

Launch the server and navigate to <http://localhost:8000/admin/>, log in with the username and password you just created for the superuser, and then add the following page within the Flatpages section:

Change flat page

URL:	<input type="text" value="/pages/about/"/>
Example: '/about/contact/'. Make sure to have leading and trailing slashes.	
Title:	<input type="text" value="About"/>
Content:	<pre>
 <p>You can add some text about yourself here. Write as much as you want. Then when you are done, just add a closing HTML paragraph tag </p> Bullet Point # 1 Bullet Point # 2 Bullet Point # 3 Bullet Point # 4
 <p>You can add a link or an email address here if you want. Again, you don't have to, but I highly recommend it. Cheers </p> <h3>Ready? Contact us!</h3></pre>

The HTML within the content portion:

```
<br>

<p>You can add some text about yourself here. Then when you are done, just add a c
losing HTML paragraph tag.</p>

<ul>
<li>Bullet Point # 1</li>
<li>Bullet Point # 2</li>
<li>Bullet Point # 3</li>
<li>Bullet Point # 4</li>
</ul>
<br/>

<p><em>You can add a link or an email address <a href="http://www.realpython.com">here</a>
here</em> if you want. Again, you don't have to, but I <strong>highly</strong> reco
mmend it. Cheers! </em></p>

<h3>Ready? Contact us!</h3>
```

Now navigate to <http://localhost:8000/pages/about/> to view the new About page.

Nice, right? Next, let's add a Contact Us page. First, your project structure should look like this:

```
└── db.sqlite3
└── django_ecommerce
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── main
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── manage.py
└── templates
    ├── base.html
    ├── flatpages
    │   └── default.html
    └── index.html
```

Contact App

Create a new app:

```
$ python manage.py startapp contact
```

Add the new app to your *settings.py* file:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',
    'django.contrib.flatpages',
    'main',
    'contact',
]
```

Set up a new model to create a database table:

```
from django.db import models
import datetime

class ContactForm(models.Model):
    name = models.CharField(max_length=150)
    email = models.EmailField(max_length=250)
    topic = models.CharField(max_length=200)
    message = models.CharField(max_length=1000)
    timestamp = models.DateTimeField(
        auto_now_add=True
    )

    def __unicode__(self):
        return self.email

    class Meta:
        ordering = ['-timestamp']
```

Which file does this code belong in? What will this code do? Let's talk it out. We are are creating a new model for our contact app, which exists within our e-commerce project. This code defines a table that will exist within our project's database, but the model itself will be used within the contact app. Therefore, we need to add this code to the `models.py` file within the "contact" directory. So now that we know how the code is interacting with our project, let's go deeper.

Does this make sense to you? The `DateTimeField` and the `ordering` within the `Meta()` class may be new. If so, take a look at the official Django docs:

- [DateTimeField](#)
- [ordering](#)

In the latter case, we are simply deviating from the default ordering and defining our own based on the data added.

Sync the database:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Add a view:

```
from .forms import ContactView
from django.contrib import messages
from django.shortcuts import render, redirect

def contact(request):
    if request.method == 'POST':
        form = ContactView(request.POST)
        if form.is_valid():
            our_form = form.save(commit=False)
            our_form.save()
            messages.add_message(
                request, messages.INFO, 'Your message has been sent. Thank you.')
    )
    return redirect('/')
else:
    form = ContactView()
return render(request, 'contact.html', {'form': form, })
```

Here, if the request is a POST (if the form is submitted, in other words) we need to process the form data. Meanwhile, if the request is anything else, then we just create a blank form. If it is a POST request and the form data is valid, then save the data, redirect the user to main page, and display a message.

NOTE: The third course dives more into the inner workings of forms, and how to properly test them to ensure data is properly being validated. Check it out!

Update your Project's *urls.py* with the following pattern:

```
from contact import views as contact_views

url(r'^contact/', contact_views.contact, name='contact'),
```

Create a new file within the "contact" directory called *forms.py*:

```
from django.forms import ModelForm
from .models import ContactForm
from django import forms

class ContactView(ModelForm):
    message = forms.CharField(widget=forms.Textarea)

    class Meta:
        model = ContactForm
        fields = ['name', 'email', 'topic', 'message']
```

Add this code to *admin.py*:

```
from django.contrib import admin
from .models import ContactForm

class ContactFormAdmin(admin.ModelAdmin):
    class Meta:
        model = ContactForm

admin.site.register(ContactForm, ContactFormAdmin)
```

Add a new file to the templates directory called *contact.html*:

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

{% block content %}

<h3><center>Contact Us</center></h3>

<form action"." method="POST" enctype="multipart/form-data">
  {% csrf_token %}
  {{ form.as_p }}
  <br>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>

{% endblock %}
```

See anything new? How about the `{% csrf_token %}` and `{{ form.as_p }}` tags? Read about the former [here](#). Why is it so important? Because you will be subject to [CSRF](#) attacks if you leave it off. Meanwhile, including the `as_p` method simply renders each for field as a separate paragraph. Check out some of the other ways you can render forms [here](#).

Update the Contact url in the `base.html` file:

```
<li><a href="{% url 'contact' %}">Contact</a></li>
```

Fire up the server and load your app. Navigate to the main page. Click the link for "contact". Test it out first with valid data, then invalid data. Then make sure that the valid data is added to the database table within the Admin page. Pretty cool. As a sanity check, open the database in the SQLite database browser and ensure that the valid data was added as a row in the correct table.

Now that you know how the contact form works, go back through this section and see if you can follow the code. Try to understand how each piece fits into the whole within the MVC structure.

Did you notice the flash message?

Your MVP!

[Home](#) [About](#) [Contact](#)

Your message has been sent. Thank you.

See if you can find the code for it in the `views.py` file. You can read more about the messages framework [here](#), which provides support for flash messages.

Updated project structure:

```
└── contact
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── forms.py
    ├── migrations
    │   ├── 0001_initial.py
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── db.sqlite3
└── django_ecommerce
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── main
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── manage.py
└── templates
    ├── base.html
    ├── contact.html
    ├── flatpages
    │   └── default.html
    └── index.html
```


User Registration with Stripe

In this last section, we will look at how to implement a payment and user authentication system. For this payment system, registration is tied to payments. So, in order to register, users must first make a payment.

We will use [Stripe](#) for payment processing.

SEE ALSO: Take a look at [this](#) brief tutorial on implementing Flask and Stripe to get a sense of how Stripe works with Python (obviously, it will be slightly different to integrate with Django).

Update the path to the `STATICFILES_DIRS` in the `settings.py` file above `STATIC_URL = '/static/'` and add the "static" directory to the root directory:

```
STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'),)
```

Create a new app:

```
$ python manage.py startapp payments
```

Add the new app to `settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',
    'django.contrib.flatpages',
    'main',
    'contact',
    'payments',
]
```

Install stripe:

```
$ pip install stripe==1.44.0
```

Update `models.py`:

```
from django.db import models
from django.contrib.auth.models import AbstractBaseUser

class User(AbstractBaseUser):
    name = models.CharField(max_length=255)
    email = models.CharField(max_length=255, unique=True)
    # password field defined in base class
    last_4_digits = models.CharField(max_length=4, blank=True, null=True)
    stripe_id = models.CharField(max_length=255)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    USERNAME_FIELD = 'email'

    def __str__(self):
        return self.email
```

SEE ALSO: This model replaces the default Django `User` model. For more on this, please visit the official [Django docs](#)

Sync the database:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Add the following import and patterns to `urls.py`:

```
from payments import views as payment_views

# user registration/authentication
url(r'^sign_in$', payment_views.sign_in, name='sign_in'),
url(r'^sign_out$', payment_views.sign_out, name='sign_out'),
url(r'^register$', payment_views.register, name='register'),
url(r'^edit$', payment_views.edit, name='edit'),
```

Add the remaining templates and all of the static files from the assets directory in the [repository](#):

- Templates: `cardform.html`, `edit.html`, `errors.html`, `field.html`, `home.html`, `register.html`, `sign_in.html`, `test.html`, `user.html`
- Static files: `application.js`, `jquery_ujs.js`, `jquery.js`, `jquery.min.js`

Take a look at the templates. These will make more sense after you see the app in action.

Now in *admin.py*:

```
from django.contrib import admin
from .models import User

class UserAdmin(admin.ModelAdmin):
    class Meta:
        model = User

admin.site.register(User, UserAdmin)
```

Add a new file called *forms.py* with the file in the assets folder of the exercises [repo](#).

Whew. That's a lot of forms. There isn't too much new happening, so let's move on. If you want, try to see how these form attributes align to the form fields in the templates.

Update *payments/views.py* with the file in the assets folder of the exercises [repo](#).

Yes, there is a lot going on here, and guess what - We're not going to go over any of it. It falls on you this time. Why? We want to see how far you're getting and how badly you really want to know what's happening in the actual code. Are you just trying to get an answer or are you taking it a step further and going through the process, working through each piece of code line by line?

Questions? Add an issue to the [Support Repo](#).

You can charge users either a one time charge or a recurring charge. This script is set up for the latter. To change to a one time charge, simply make the following changes to the file:

```
if form.is_valid():
    # update based on your billing method (subscription vs one time)
    # customer = stripe.Customer.create(
    #     email=form.cleaned_data['email'],
    #     description=form.cleaned_data['name'],
    #     card=form.cleaned_data['stripe_token'],
    #     plan="gold",
    # )
    # )
    customer = stripe.Charge.create(
        description = form.cleaned_data['email'],
        card = form.cleaned_data['stripe_token'],
        amount="5000",
        currency="usd"
    )

```

Yes, this is a bit of a hack. We'll show you an elegant way of doing this in the next course.

Your project structure should now look like this:

```
└── contact
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── forms.py
    ├── migrations
    │   ├── 0001_initial.py
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── db.sqlite3
└── django_ecommerce
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── main
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── manage.py
└── payments
```

```
|- __init__.py
|- admin.py
|- apps.py
|- forms.py
|- migrations
|   |- 0001_initial.py
|   |- __init__.py
|- models.py
|- tests.py
|- views.py
|- static
|   |- application.js
|   |- bootstrap.css
|   |- jquery.js
|   |- jquery.min.js
|   |- jquery_ujs.js
|- templates
|   |- base.html
|   |- cardform.html
|   |- contact.html
|   |- edit.html
|   |- errors.html
|   |- field.html
|   |- flatpages
|       |- default.html
|   |- home.html
|   |- index.html
|   |- register.html
|   |- sign_in.html
|   |- test.html
|   |- user.html
```

Update *main/views.py*:

```
from django.shortcuts import render
from payments.models import User

def index(request):
    uid = request.session.get('user')
    if uid is None:
        return render(request, 'index.html')
    else:
        return render(
            request,
            'user.html',
            {'user': User.objects.get(pk=uid)})
)
```

Update the `base.html` template with `base_version_2.html` from the assets folder in the [repo](#).

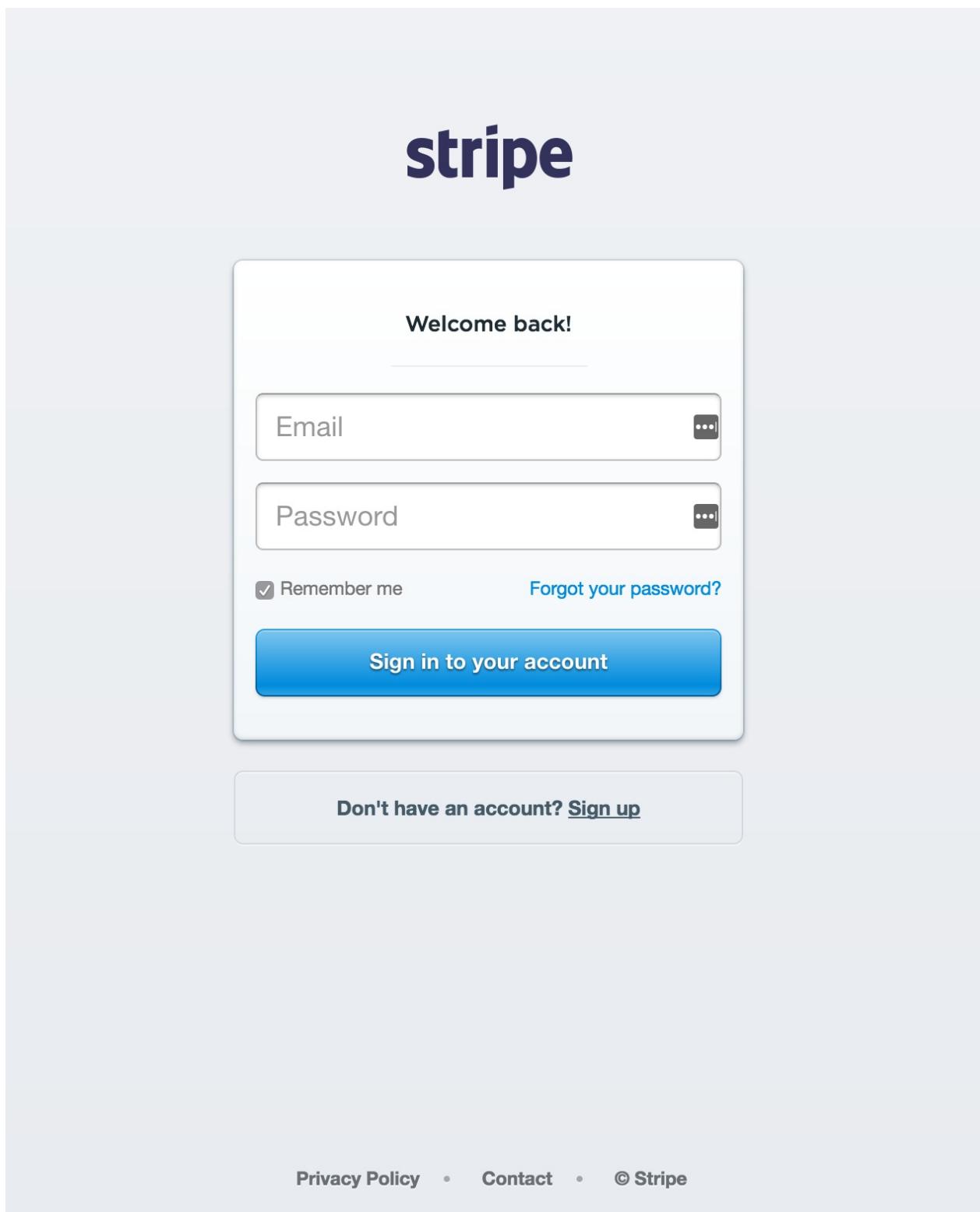
Update the large button in `index.html`:

```
<a class="btn btn-large btn-success" href="/register">Register today to get started!</a><br/><br/><br/>
```

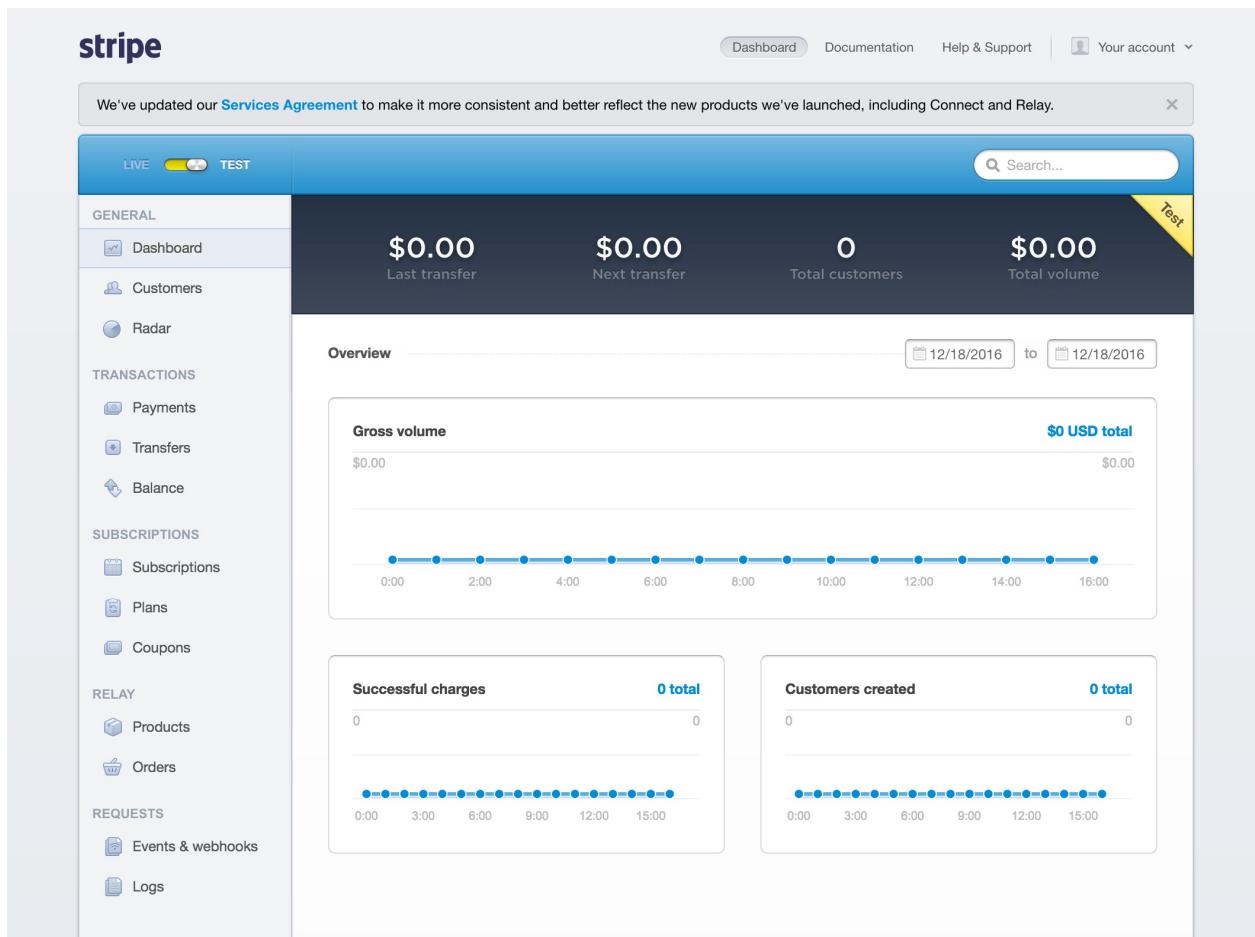
Stripe

Before we can start charging customers, we must grab the API keys and add a subscription plan, which we called `plan="gold"` back in our `views.py` file. This is the plan ID.

Go to the Stripe test Dashboard at <https://dashboard.stripe.com/test/dashboard>.



The first time you go to this URL, it will ask you to log in. Don't. Instead, click the link to sign up, then on the sign up page, click the link to skip this step. Finally, on the next view, click the link to 'go straight to your dashboard'. You should now be taken to the dashboard page.



Now, you should be able to access the keys, by going to this URL:

<https://dashboard.stripe.com/account/apikeys>.

Add the keys to *settings.py*:

```
STRIPE_SECRET = 'GET YOUR OWN'  
STRIPE_PUBLISHABLE = 'GET YOUR OWN'
```

Test payment

Go ahead and test this out using the test keys. Fire up the server. Navigate to the main page, then click Register.

Fill everything out. Make sure to use the credit card number 4242424242424242 and any 3 digits for the CVC code. Use any date in the future for the expiration date.

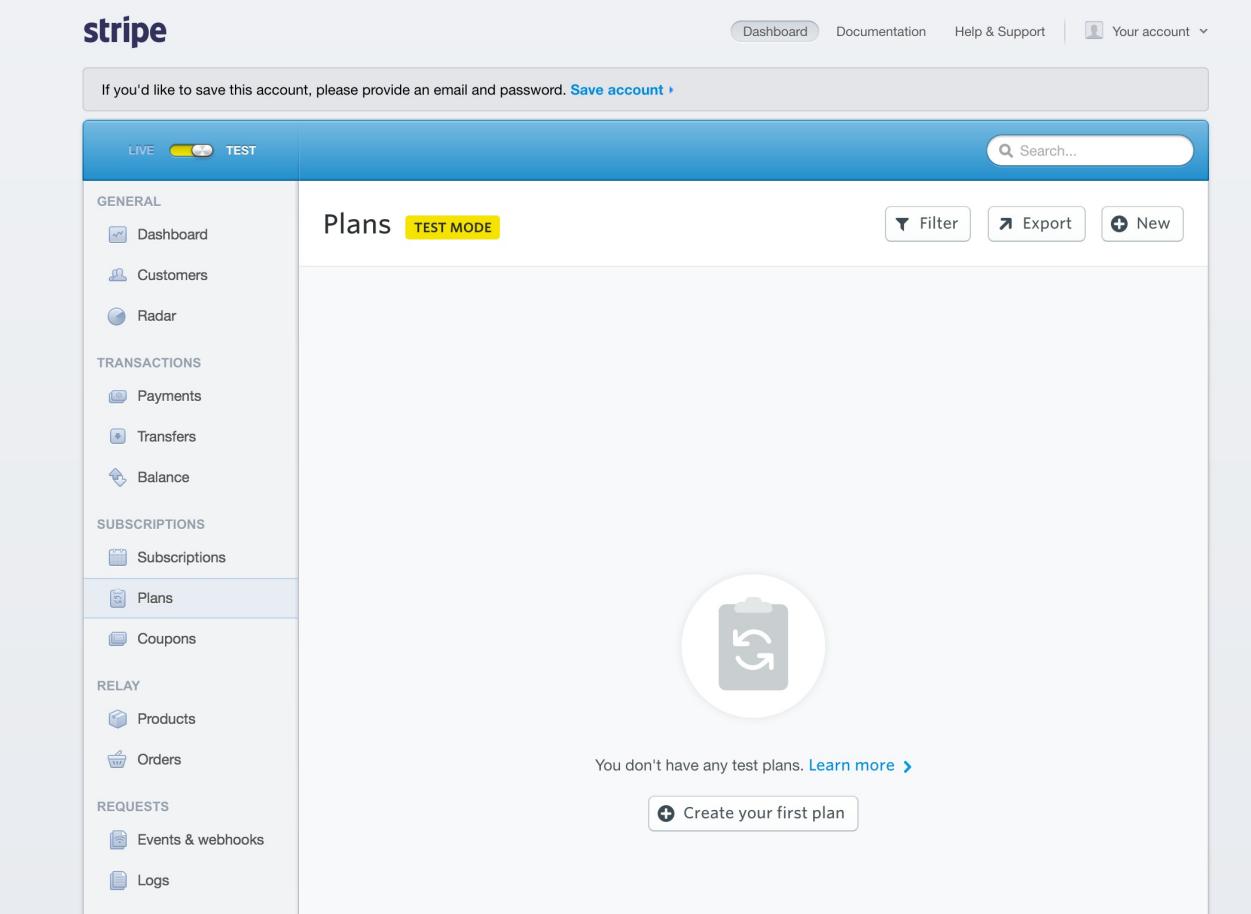
Click submit. You should get an error that says, "No such plan: gold". If you go back to the view, you will see this code indicating that this is looking for a Stripe plan with the id of 'gold':

```
customer = stripe.Customer.create(  
    email = form.cleaned_data['email'],  
    description = form.cleaned_data['name'],  
    card = form.cleaned_data['stripe_token'],  
    plan="gold",  
)
```

NOTE: As of April of 2016, Stripe has [adopted more secure protocols](#). In this chapter the TLS upgrade is likely to affect your progress with integrating Stripe into your application. But have no fear as this is a great learning opportunity, although I must admit, a bit tricky. It requires you to update your 'OpenSSL' version to 1.0.1 or greater (currently 1.0.1j) and this newer version must then be used by Python3 within your development environment. This setup process will be similar on most computers but exactly the same. Here are a bunch of tutorials and posts to help you on this journey:

- <https://support.stripe.com/questions/how-do-i-upgrade-my-openssl-to-support-tls-1-2>
- <https://support.stripe.com/questions/how-do-i-upgrade-my-stripe-integration-from-tls-1-0-to-tls-1-2#python> Mac OS X
- <http://apple.stackexchange.com/questions/126830/how-to-upgrade-openssl-in-os-x>
- <https://github.com/Homebrew/legacy-homebrew/issues/14497>

Let's create that plan now. Kill the server. Then, back on Stripe, click *Plans* and then *Create your first plan*. Enter the following data into the form:



If you'd like to save this account, please provide an email and password. [Save account](#)

LIVE TEST

Search...

GENERAL

- Dashboard
- Customers
- Radar

TRANSACTIONS

- Payments
- Transfers
- Balance

SUBSCRIPTIONS

- Subscriptions
- Plans**
- Coupons

RELAY

- Products
- Orders

REQUESTS

- Events & webhooks
- Logs

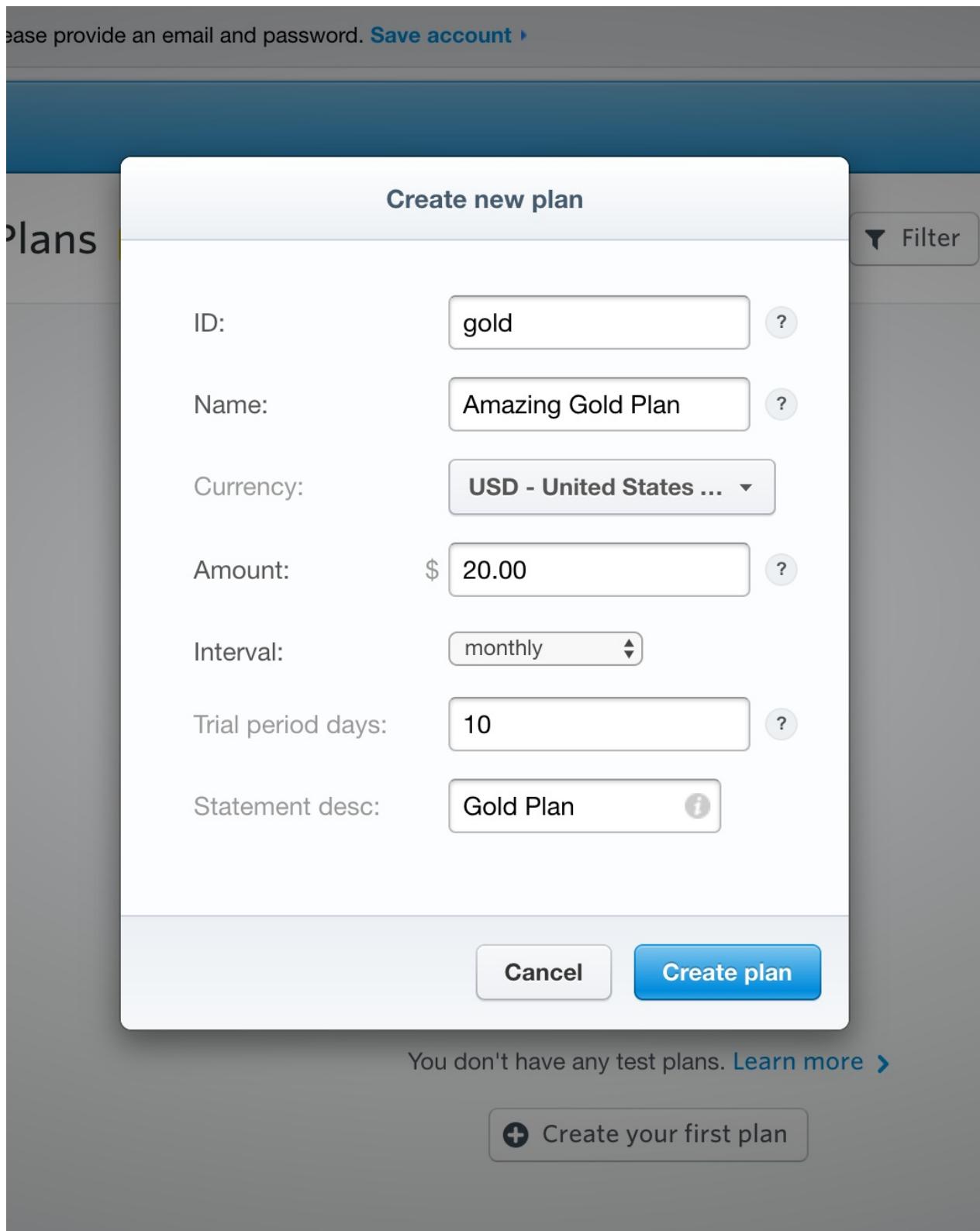
Plans TEST MODE

Filter Export New

You don't have any test plans. [Learn more](#)

Create your first plan

- ID: gold
- Name: Amazing Gold Plan
- Amount: 20
- Interval: monthly
- Trial Period: 10
- Statement Description: Gold Plan



Click *Create plan*.

SEE ALSO: Refer to the [Stripe documentation](#) and [API reference docs](#) for more info.

Go back to your app. Now you should be able to register a new user.

Your MVP!

[Home](#) [About](#) [Contact](#)

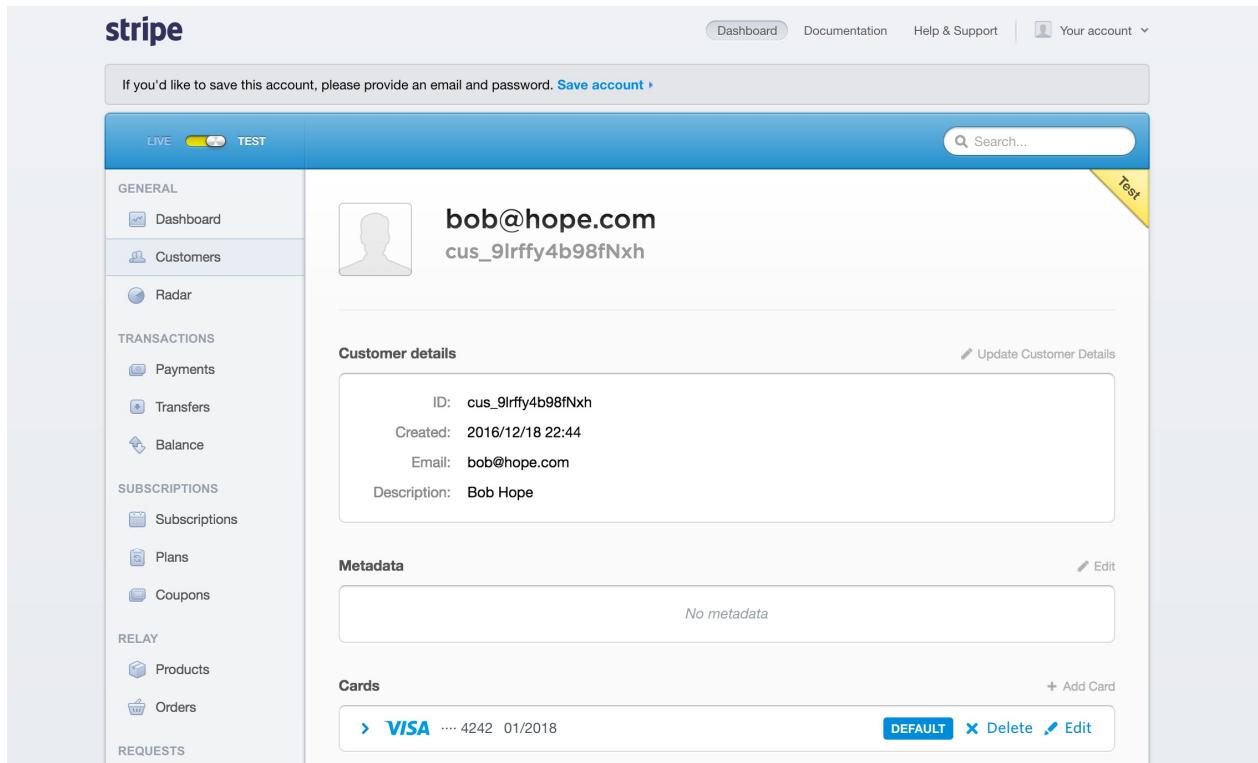
Member's Only Page

Welcome Bob Hope.

Click [here](#) to make changes to your credit card.

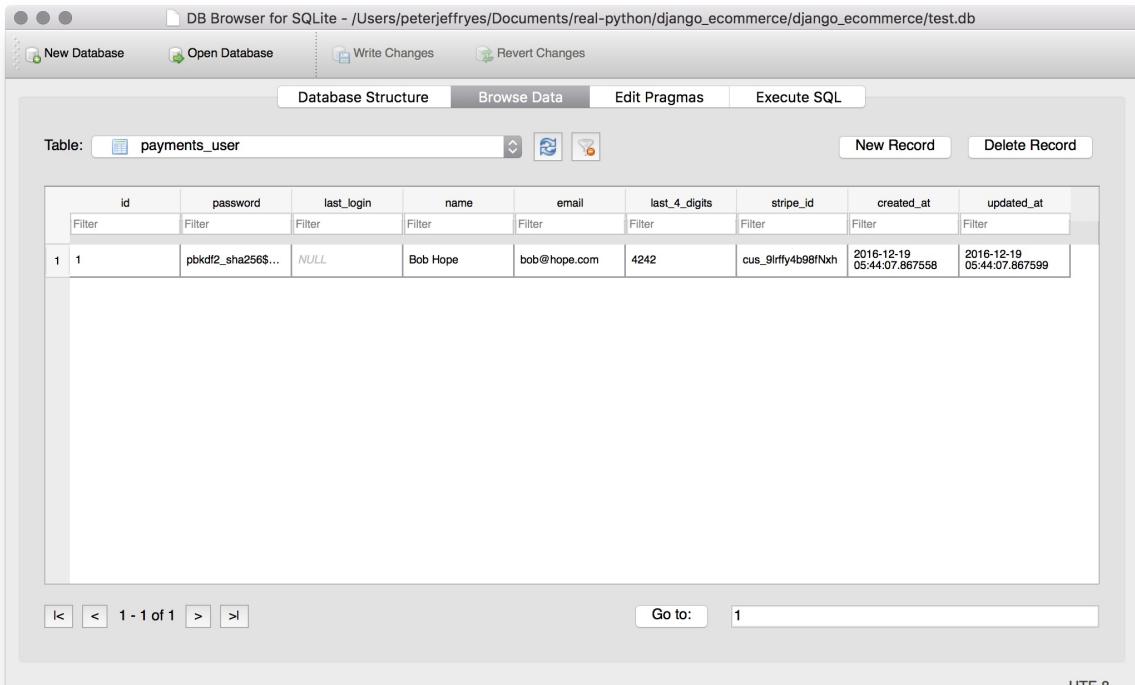
Copyright © 2014 Your MVP. Developed by Your Name Here. Powered by Django.

Finally, after you process the test/dummy payment, make sure the subscription was added to the Stripe dashboard.



The screenshot shows the Stripe dashboard interface. On the left, a sidebar menu includes options like LIVE, TEST, Dashboard, Customers, Radar, Payments, Transfers, Balance, Subscriptions, Plans, Coupons, Products, Orders, and Requests. The main content area displays a customer profile for 'bob@hope.com' (cus_9lrfy4b98fNxh). The profile includes a placeholder user icon, the email 'bob@hope.com', and the ID 'cus_9lrfy4b98fNxh'. Below this, the 'Customer details' section shows the ID, creation date (2016/12/18 22:44), email, and description ('Bob Hope'). The 'Metadata' section indicates 'No metadata'. The 'Cards' section shows a single card entry: a VISA card ending in 4242, with options to set it as 'DEFAULT', 'Delete', or 'Edit'.

Take a look at the 'payments_users' collection in the SQLiteBrowser to make sure your customer info saved:



The screenshot shows the DB Browser for SQLite interface. The database is set to 'payments_user'. The table structure is displayed with columns: id, password, last_login, name, email, last_4_digits, stripe_id, created_at, and updated_at. A single record is present with the following data:

	id	password	last_login	name	email	last_4_digits	stripe_id	created_at	updated_at
1	1	pbkdf2_sha256\$...	NULL	Bob Hope	bob@hope.com	4242	cus_9irfty4b98fNxh	2016-12-19 05:44:07.867558	2016-12-19 05:44:07.867599

Congrats! You know have a basic application up and running.

It's up to you to figure out what type of product or service you are offering the user.

You can update a user's credit card info within the Member's only page. Try this. Change the CVC code and expiration data. Then, back on Stripe, click *Customers* and then the customer. You should see an event showing that the changes were made. If you click on the actual event, you can see the raw JSON sent that shows the changes you made.

Next Steps

Think about some of the other things that you'd want to allow a user to do. Update their plan - if you had separate plan tiers, of course. Cancel. Maybe they could get a monthly credit if they invite a friend to your app.

In the next course, we will show you how to do this and more. See you then. Cheers!

Appendix A: Installing Python

Let's get [Python 3.6](#) installed!

Check Current Version

Mac and Linux

All Mac OS X versions since 10.4 and most Linux distributions come pre-installed with the latest version of Python 2.7.x. You can view the version by opening the terminal and typing `python` to enter the Python interpreter. The output will look something like this:

```
$ python
Python 2.7.12 (default, Oct 11 2016, 05:24:00)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOTE: If you have a version older than 3.6, please download the latest version below.

Install Python

Choose your Operating system...

Mac

You need a Python version 3.5+. So, if you need to download a new version, download the latest [installer](#) for version 3.6.0.

Once downloaded, double-click the file to install.

Linux

If you are using Ubuntu, Linux Mint, or another Debian-based system, enter the following command in your terminal to install Python:

```
$ sudo apt-get install python3.6
```

Or you can download the tarball directly from the official Python [website](#). Once downloaded, run the following commands:

```
$ tar -zxvf [mytarball.tar.gz]
$ ./configure
$ make
$ sudo make install
```

NOTE: If you have problems or have a different Linux distribution, you can always use your package manager or just do a Google search for how to install Python on your particular Linux distribution.

Windows

Download

Start by downloading Python 3.6.0 from the official Python [website](#). The Windows version is distributed as a MSI package. Once downloaded, double-click to start the installer. Follow the installer instructions to completion. By default this will install Python to `C:\Python35`.

NOTE: You may need to be logged in as the administrator to run the install.

Test

To test this install open your command prompt, which should open to the C:prompt, `c:/>`, then type:

```
\Python35\python.exe
```

And press enter. You should see something like:

```
Python 3.6.0 (v3.6.0:37a07cee5969, ... ) on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Yay! You just started Python!

NOTE: The `>>>` indicates that you are at the Python interpreter (or prompt) where you can run Python code interactively.

To exit the Python prompt, type:

```
exit()
```

Then press Enter. This will take you back to the C:prompt.

Path

You also need to add Python to your PATH environmental variables, so when you want to run a Python script, you do not have to type the full path each and every time, as this is quite tedious. In other words, after adding Python to the PATH, we will be able to simply type `python` in the command prompt rather than `\Python35\python.exe`.

Since you downloaded Python version 3.6, you need to add the add the following directories to your PATH:

- `C:\Python36\`
- `C:\Python36\Scripts\`
- `C:\PYTH0N36\DLLs\`
- `C:\PYTH0N36\LIB\`

Open your power shell and run the following statement:

```
[Environment]::SetEnvironmentVariable("Path",
    "$env:Path;C:\Python35\;C:\Python36\Scripts\;
    C:\PYTHON36\DLLs\;C:\PYTHON36\LIB\;", "User")
```

That's it.

Video

Watch the video [here](#) for assistance. Note: Even though this is an older version of Python the steps are the same.

Verify Install

Test this new install by opening a *new* terminal, then type `python`. You should see the same output as before except the version number should now be 3.6 (or whatever the latest version of Python is):

```
$ python
Python 3.6 (default, Feb 28 2017, 05:05:28)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOTE: You may need to run `python3` instead of `python` if you have multiple versions of Python installed.

Congrats! You have Python installed and configured.

Appendix B: Supplementary Materials

Working with FTP

There are a number of different ways to exchange files over the Internet. One of the more popular ways is to connect to a FTP server to download and upload files. FTP (File Transfer Protocol) is used for file exchange over the Internet. Much like HTTP and SMTP - which are used for exchanging web pages and email across the Internet, respectively - FTP uses the TCP/IP protocols for transferring data.

In most cases, FTP is used to either upload a file (such as a web page) to a remote server, or download a file from a server. In this lesson, we will access an FTP server to view the main directory listing, upload a file, and then download a file.

```
import ftplib

server = ''
username = ''
password = ''

# Initialize and pass in FTP URL and login credentials (if applicable)
ftp = ftplib.FTP(host=server, user=username, passwd=password)

# Create a list to receive the data
data = []

# Append the directories and files to the list
ftp.dir(data.append)

# Close the connection
ftp.quit()

# Print out the directories and files, line by line
for l in data:
    print(l)
```

Save the file. *DO NOT* run it just yet.

What's going on here?

We imported the `ftplib` library, which provides Python the dependencies it needs to access remote servers, files, and directories. We then defined variables for the remote server and the login credentials to initialize the FTP connection. You can leave the username and password empty if you are connecting to an anonymous FTP server

(which usually doesn't require a username or password). Next, we created a list to receive the directory listing, and used the `dir` command to append data to the list. Finally, we disconnected from the server and then printed the directories, line by line, using a `for` loop.

Let's test this out with a public ftp site, using these server and login credentials:

```
server = 'ftp.debian.org'  
username = 'anonymous'  
password = 'anonymous'
```

Also add the following line just after you create the list:

```
# change into the debian directory  
ftp.cwd('debian')
```

Keep everything else the same, and save the file again. Now you can run it. If done correctly your output should look like this:

```
-rw-rw-r-- 1 1176 1176 1062 Sep 17 09:53 README  
-rw-rw-r-- 1 1176 1176 1290 Jun 26 2010 README.CD-manufacture  
-rw-rw-r-- 1 1176 1176 2590 Sep 17 09:53 README.html  
-rw-r--r-- 1 1176 1176 190914 Dec 07 19:52 README.mirrors.html  
-rw-r--r-- 1 1176 1176 103482 Dec 07 19:52 README.mirrors.txt  
drwxr-sr-x 18 1176 1176 4096 Sep 17 09:55 dists  
drwxr-sr-x 4 1176 1176 4096 Dec 09 07:52 doc  
-rw-r--r-- 1 1176 1176 143221 Dec 09 08:55 extrafiles  
drwxr-sr-x 3 1176 1176 4096 Jul 27 2015 indices  
-rw-r--r-- 1 1176 1176 12014554 Dec 09 08:48 ls-1R.gz  
drwxr-sr-x 5 1176 1176 4096 Dec 19 2000 pool  
drwxr-sr-x 4 1176 1176 4096 Nov 17 2008 project  
drwxr-xr-x 3 1176 1176 4096 Oct 10 2012 tools  
drwxr-xr-x 6 1176 1176 4096 May 21 2016 zzz-dists
```

Next, let's take a look at how to download a file from a FTP server.

```
import ftplib
import sys

server = 'ftp.debian.org'
username = 'anonymous'
password = 'anonymous'

# Defines the name of the file for download
file_name = sys.argv[1]

# Initialize and pass in FTP URL and login credentials (if applicable)
ftp = ftplib.FTP(host=server, user=username, passwd=password)

ftp.cwd('debian')

# Create a local file with the same name as the remote file
with open(file_name, "wb") as f:

    # Write the contents of the remote file to the local file
    ftp.retrbinary("RETR " + file_name, f.write)

# Closes the connection
ftp.quit()
```

When you run the file, make sure you specify a file name from the remote server on the command line. You can use any of the files you saw in the above script when we outputted them to the screen.

For example:

```
$ python3 ftp_download.py README
```

Finally, let's take a look at how to upload a file to a FTP Server.

```
import ftplib
import sys

server = ''
username = ''
password = ''

# Defines the name of the file for upload
file_name = sys.argv[1]

# Initialize and pass in FTP URL and login credentials (if applicable)
ftp = ftplib.FTP(host=server, user=username, passwd=password)

# Opens the local file to upload
with open(file_name, "rb") as f:

    # Write the contents of the local file to the remote file
    ftp.storbinary("STOR " + file_name, f)

# Closes the connection
ftp.quit()
```

Save the file. *DO NOT* run it.

Unfortunately, the public FTP site we have been using does not allow uploads. You will have to use your own server to test. Many free hosting services offer FTP access. You can set one up in less than fifteen minutes. Just search Google for "free hosting with ftp" to find a free hosting service. One good example is <http://www.0fees.net>.

After you set up your FTP Server, update the server name and login credentials in the above script, save the file, and then run it. *Again, specify the filename as one of the command line arguments. It should be a file found on your local directory.*

For example:

```
$ python3 ftp_upload.py uploadme.txt
```

Check to ensure that the file has been uploaded to the remote directory

NOTE: It's best to send files in binary mode, "rb", as this mode sends the raw bytes of the file. Thus, the file is transferred in its exact original form.

Homework

Write a script to navigate to a specific directory, upload a file, and then run a directory listing of that directory to see the newly uploaded file.

Working with SFTP

SFTP (Secure File Transfer Protocol) is used for securely exchanging files over the Internet. From an end-user's perspective it is very much like FTP, except that data is transported over a secure channel. To use SFTP you have to install the `pysftp` library. There are other custom libraries for SFTP but this one is good if you want to run SFTP commands with minimal configuration.

Install `pysftp` :

```
$ pip3 install pysftp
```

Now, let's try to list the directory contents from a remote server via SFTP:

```
import pysftp

server = ''
username = ''
password = ''

# Initialize and pass in SFTP URL and login credentials (if applicable)
sftp = pysftp.Connection(host=server, username=username, password=password)

# Get the directory and file listing
data = sftp.listdir()

# Closes the connection
sftp.close()

# Prints out the directories and files, line by line
for l in data:
    print(l)
```

Save the file. Once again, *DO NOT* run it.

To test the code, you will have to use your own remote server that supports SFTP. Unfortunately, most of the free hosting services do not offer SFTP access. But don't worry, there are free sites that offer *free shell accounts* which normally include SFTP access. Just search Google for "free shell accounts".

After you set up your SFTP Server, update the server name and login credentials in the above script, save the file, and then run it. If done correctly, all the files and directories of the current directory of your remote server will be displayed.

Next, let's take a look at how to download a file from an SFTP server:

```
import pysftp
import sys

server = ''
username = ''
password = ''

# Defines the name of the file for download
file_name = sys.argv[1]

# Initialize and pass in SFTP URL and login credentials (if applicable)
sftp = pysftp.Connection(host=server, username=username, password=password)

# Download the file from the remote server
sftp.get(file_name)

# Closes the connection
sftp.close()
```

When you run it, make sure you specify a file name from the remote server on the command line. You can use any of the files you saw in the above script when you outputted them to the screen.

For example:

```
$ python test_sftp_download.py remotefile.csv
```

Finally, let's take a look at how to upload a file to an SFTP Server:

```
# ex_appendixB.2c.py - SFTP File Upload

import pysftp
import sys

server = ''
username = ''
password = 'p@'

# Defines the name of the file for upload
file_name = sys.argv[1]

# Initialize and pass in SFTP URL and login credentials (if applicable)
sftp = pysftp.Connection(host=server, username=username, password=password)

# Upload the file to the remote server
sftp.put(file_name)

# Closes the connection
sftp.close()
```

When you run this, make sure you specify a file name from your local directory on the command line.

For example:

```
$ python3 test_sftp_download.py remotefile.csv
```

Check to ensure that the file has been uploaded to the remote directory

Sending and Receiving Email

SMTP (Simple Mail Transfer Protocol) is the protocol that handles sending and routing email between mail servers. The `smtplib` library from the Python standard library is used to define the SMPT client session object implementation, which is used to send mail through Unix mail servers. MIME is an Internet standard used for building the various parts of emails, such as "From", "To", "Subject", and so forth. We will be using that library as well.

Sending mail is simple.

```
# Sending Email via SMTP (part 1)

import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

# inputs for from, to, subject and body text
fromaddr = input("Sender's email: ")
toaddr = input('To: ')
sub = input('Subject: ')
text = input('Body: ')

# email account info from where we'll be sending the email from
smtp_host = 'smtp.mail.com'
smtp_port = '###'
user = 'username'
password = 'password'

# parts of the actual email
msg = MIMEMultipart()
msg['From'] = fromaddr
msg['To'] = toaddr
msg['Subject'] = sub
msg.attach(MIMEText(text))

# connect to the server
server = smtplib.SMTP(smtp_host, smtp_port)

# initiate communication with server
server.ehlo()

# use encryption
server.starttls()

# login to the server
server.login(user, password)

# send the email
server.sendmail(fromaddr, toaddr, msg.as_string())

# close the connection
server.quit()
```

Save. *DO NOT* run.

Since this code is pretty self-explanatory (follow along with the inline code comments), go ahead and update the the following variables: `smtp_host` , `smtp_port` , `user` , `password` to match your email account's SMTP info and login credentials you wish to

send from.

Example:

```
# email account info from where we'll be sending the email from
smtp_host = 'smtp.gmail.com'
smtp_port = 587
user = 'hermanmu@gmail.com'
password = "it's a secret - sorry"
```

Try using a Gmail account to send and receive from the same address at first to test it out, and then send a message from Gmail to a different email account, on a different email service. Once complete, run the file. As long as you don't get an error, the email should have sent correctly. Check your email to make sure.

Before moving on, let's clean up the code a bit:

```
# ex_appendixB.3b.py - Sending Email via SMTP (part 2)

import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

def mail(fromaddr, toaddr, sub, text, smtp_host, smtp_port, user, password):

    # parts of the actual email
    msg = MIMEMultipart()
    msg['From'] = fromaddr
    msg['To'] = toaddr
    msg['Subject'] = sub
    msg.attach(MIMEText(text))

    # connect to the server
    server = smtplib.SMTP(smtp_host, smtp_port)

    # initiate communication with server
    server.ehlo()

    # use encryption
    server.starttls()

    # login to the server
    server.login(user, password)

    # send the email
    server.sendmail(fromaddr, toaddr, msg.as_string())

    server.quit()

if __name__ == '__main__':
    fromaddr = 'hermanmu@gmail.com'
    toaddr = 'hermanmu@gmail.com'
    subject = 'test'
    body_text = 'hear me?'
    smtp_host = 'smtp.gmail.com'
    smtp_port = '587'
    user = 'hermanmu@gmail.com'
    password = "it's a secret"

    mail(fromaddr, toaddr, subject, body_text, smtp_host, smtp_port, user, password)
```

Meanwhile, IMAP (Internet Message Access Protocol) is the Internet standard for receiving email on a remote mail server. Python provides the `imaplib` library as part of the standard library which is used to define the IMAP client session implementation,

used for accessing email. Essentially, we will set up our own mail server.

```
# Receiving Email via IMAPLIB

import imaplib

# email account info from where we'll be sending the email from
imap_host = 'imap.gmail.com'
imap_port = '993'
user = 'hermanmu@gmail.com'
password = "It's a secret - sorry!"

# login to the mail server
server = imaplib.IMAP4_SSL(imap_host, imap_port)
server.login(user, password)

# select the inbox
status, num = server.select('Inbox')

#fetch the email and the information you wish to display
status, data = server.fetch(num[0], '(BODY[TEXT])')

# print the results
print(data[0][1].decode("utf-8", "ignore"))
server.close()
server.logout()
```

Notice how we used the same Gmail account from the last example. Make sure you tailor this to your own account settings. Essentially, this code is used to read the most recent email in the Inbox. This just so happened to be the email I sent myself in the last example.

Another thing I would like to point out is this line of code:

```
status, num = server.select('Inbox')
```

Here we know that `server.select` returns a tuple of 2 values. This syntax unpacks the tuple and assigns values to the variables in the order they are listed.

Ok so now, if done correctly, you should see something like this:

```
-----7968962300591266385==  
Content-Type: text/plain; charset="us-ascii"  
MIME-Version: 1.0  
Content-Transfer-Encoding: 7bit  
  
Test Python - what up!  
-----7968962300591266385==
```

Test Python - what up! is the body of my email.

Also, in the program `num[0]` it specifies the message we wish to view, while `(BODY[TEXT])` displays the information from the email.

Homework

See if you can figure out how to use a loop to read the first 10 messages in your inbox.

Acknowledgements

Writing is an intense, solitary activity that requires discipline and repetition. Although much of what happens in that process is still a mystery to me, I know that the my friends and family have played a huge role in the development of this course. I am immensely grateful to all those in my life for providing feedback, pushing me when I needed to be pushed, and just listening when I needed silent support.

At times I ignored many people close to me, despite their continued support. You know who you are. I promise I will do my best to make up for it.

For those who wish to write, know that it can be a painful process for those around you. They make just as many sacrifices as you do - if not more. Be mindful of this. Take the time to be in the moment with those people, in any way that you can. You and your work will benefit from this.

Thank you

First and foremost, I'd like to thank Fletcher and Jeremy, authors of the other Real Python courses, for believing in me even when I did not believe in myself. They both are talented developers and natural leaders; I'm also proud to call them friends. Thanks to all my close friends and family (Mom, Dad, Jeff) for all your support and kindness. Derek, Josh, Danielle, Richard, Lily, John (all three of you), Marcy, and Travis - each of you helped in a very special way that I am only beginning to understand.

Thank you also to the immense support from the Python community. Despite not knowing much about me or my abilities, you welcomed me, supported me, and shaped me into a much better programmer. I only hope that I can give back as much as you have given me.

Thanks to all who read through drafts, helping to shape this course into something accurate, readable, and, most importantly, useful. Nina, you are a wonderful technical writer and editor. Stay true to your passions.

Thank you Massimo, Shea, and Mic. You are amazing.

For those who don't know, this course started as a Kickstarter. To all my original backers and supporters: You have lead me as much as I hope I am now leading you. Keep asking for more. This is your course.

Finally, thank you to a very popular yet terrible API that forced me to develop my own solution to a problem, pushing me back into the world of software development. Permanently.

About the Author

[Michael](#) is a lifelong learner. Formally educated in computer science, philosophy, business, and information science, he continues to challenge himself by learning new languages and reading *Infinite Jest* over and over again. He's been hacking away since developing his first project - a video game enthusiast website - back in 1999.

Python is his tool of choice. He's founded and co-founded several startups and has written extensively on his experiences.

He loves libraries and other mediums that provide publicly available data. When not staring at a computer screen, he enjoys running, writing flash fiction, and making people feel uncomfortable with his [dance moves](#).

About the Editor

[Massimo Di Pierro](#) is an associate professor at the School of Computing of DePaul University in Chicago, where he directs the Master's program in Computational Finance. He also teaches courses on various topics, including web frameworks, network programming, computer security, scientific computing, and parallel programming.

Massimo has a PhD in High Energy Theoretical Physics from the University of Southampton (UK), and he has previously worked as an associate researcher for Fermi National Accelerator Laboratory. Massimo is the author of a book on `web2py`, and more than 50 publications in the fields of Physics and Computational Finance, and he has contributed to many open source projects.

He started the `web2py` project in 2007, and is currently the lead developer.

Python 2 to 3

Thanks to [Pete Jeffryes](#), in the final months of 2016 the entirety of this book was revised to make use of the newest versions of all relied upon frameworks and dependencies including, the shift from Python 2 to 3. Pete is a developer out of Denver, Colorado who has always enjoyed the art of language learning from jazz and classical music to Mandarin Chinese, and now the languages of software development. His was first exposed to Python through an MIT OpenCourseWare offering in 2014 and he has been hooked ever since.