



Part 2: Web Development with Python

Real Python Part 2: Web Development with Python

Michael Herman

Contents

1	Preface	10
	Acknowledgements	10
	Thank you	11
	About the Author	12
	About the Editor	13
	License	14
2	Introduction	15
	Why Python?	16
	Who should take this Course?	17
	How to use this Course	18
	Course Repository	19
	Conventions	20
	Errata	22
3	Getting Started	23
	Python Review	23
	Development Environments	25
	Installing SQLite	29
	Installing easy_install and pip	30
	Installing virtualenv	32
	Web Browsers	34

Version Control	36
4 Interlude: Modern Web Development	41
Overview	41
Front-end, Back-end, and Middleware	42
Model-View-Controller (MVC)	43
5 Flask: Quick Start	45
Overview	46
Installation	47
Hello World	48
Flow of the App	51
Dynamic Routes	52
Response Object	54
Debug Mode	56
6 Interlude: Database Programming	58
SQL and SQLite Basics	59
Creating Tables	61
Inserting Data	64
Searching	69
Updating and Deleting	71
Working with Multiple Tables	73
SQL Functions	77
Example Application	79
SQL Summary	83
Summary	84

7	Flask Blog App	85
	Project Structure	86
	Model	88
	Controller	89
	Views	91
	Templates	93
	Run the server!	95
	User Login	96
	Sessions and Login_required Decorator	100
	Show Posts	105
	Add Posts	107
	Style	109
	Conclusion	111
8	Interlude: Debugging in Python	112
	Workflow	113
	Post Mortem Debugging	120
9	Flask: FlaskTaskr, Part 1 - Quick Start	122
	Overview	122
	Setup	123
	Configuration	124
	Database	125
	Controller	127
	Templates and Styles	129
	Test	134
	Tasks	136
	Add, Update, and Delete Tasks	138
	Tasks Template	140

Add Tasks form	144
Test	146
10 Flask: FlaskTaskr, Part 2 - SQLAlchemy and User Management	148
Database Management	150
User Registration	158
User Login/Authentication	164
Database Relationships	166
Managing Sessions	175
11 Flask: FlaskTaskr, Part 3 - Error Handling and Unit Testing	178
Error Handling	180
Unit Testing	191
12 Interlude: Introduction to HTML and CSS	205
HTML	206
CSS	209
Chrome Developer Tools	213
13 Flask: FlaskTaskr, Part 4 - Styles, Test Coverage, and Permissions	214
Templates and Styling	216
Test Coverage	226
Nose Testing Framework	228
Permissions	230
14 Flask: FlaskTaskr, Part 5 - Blueprints	240
What are Blueprints?	242
Example Code	243
Refactoring our app	245

15 Flask: FlaskTaskr, Part 6 - New features!	263
New Features	265
Password Hashing	271
Custom Error Pages	275
Error Logging	282
Deployment Options	285
Automated Deployments	288
Building a REST API	293
Boilerplate Template and Workflow	305
16 Flask: FlaskTaskr, Part 7: Continuous Integration and Delivery	309
Workflow	310
Continuous Integration Tools	311
Travis CI Setup	312
Intermission	315
Feature Branch Workflow	316
Fabric	319
Recap	320
Conclusion	321
17 Flask: Behavior-Driven Development with Behave	323
Behavior-Driven Development	324
Project Setup	327
Introduction to Behave	328
Feature Files	329
First Feature	330
Environment Control	332
Next steps	334
Login and Logout	336

Second Feature	339
Third Feature	343
Update Steps	348
Conclusion	350
18 Interlude: Web Frameworks, Compared	351
Overview	351
Popular Frameworks	353
Components	354
What does this all mean?	355
19 web2py: Quick Start	356
Overview	356
Installation	358
Hello World	359
Deploying on PythonAnywhere	364
seconds2minutes App	366
20 Interlude: APIs	369
Introduction	369
Retrieving Web Pages	371
Web Services Defined	374
Working with XML	378
Working with JSON	382
Working with Web Services	387
My API Films	398
21 web2py: Sentiment Analysis	402
Sentiment Analysis	402
Sentiment Analysis Expanded	414

Movie Suggester	426
Blog App	434
22 web2py: py2manager	437
Introduction	437
Setup	439
Database	441
URL Routing	449
Initial Views	451
Profile Page	455
Add Projects	457
Add Companies	459
Homepage	460
More Grids	462
Notes	464
Error Handling	467
Final Word	468
23 Interlude: Web Scrapping and Crawling	469
Libraries	472
HackerNews (scrapy.Spider)	473
Scrapy Shell	477
Wikipedia (scrapy.Spider)	479
Socrata (CrawlSpider and Item Pipeline)	483
Web Interaction	489
24 web2py: REST Redux	492
Introduction	492
Basic REST	494
Advanced REST	498

25 Django: Quick Start	503
Overview	503
Installation	506
Hello, World!	507
26 Interlude: Introduction to JavaScript and jQuery	517
Handling the Event	519
Append the text to the DOM	521
Remove text from the DOM	523
27 Bloggy: A blog app	525
Setup	526
Model	527
Setup an App	528
Django Shell	531
Unit Tests for Models	536
Django Admin	538
Custom Admin View	539
Templates and Views	540
Friendly Views	546
Django Migrations	548
View Counts	552
Styles	553
Popular Posts	557
Forms	562
Even Friendlier Views	571
Stretch Goals	578
28 Django Workflow	579

29 Bloggy Redux: Introducing Blongo	582
MongoDB	583
Talking to Mongo	584
Django Setup	586
Setup an App	587
Add Data	588
Update Project	589
Test	593
Test Script	594
Conclusion	596
 30 Django: Ecommerce Site	 597
Overview	597
Rapid Web Development	598
Prototyping	599
Setup your Project and App	601
Add a landing page	604
Bootstrap	612
Add an about page	613
Contact App	617
User Registration with Stripe	622
 31 Appendix A: Installing Python	 640
Windows	641
Mac OS X	644
Linux	645
 32 Appendix B: Supplementary Materials	 646
Working with FTP	646
Working with SFTP	651
Sending and Receiving Email	654

Chapter 1

Preface

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 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 continuing 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 Piero](#) 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.

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.

Chapter 2

Introduction

This is not a reference book. Instead, I'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 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 on the [forum](#). 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, I urge you to approach this course with a beginner's mind. The best way to learn this material is by challenging yourself. Take my examples further. Find errors in my 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 are unclear or lack clarity, please contact me at info@realpython.com so I 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 (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, and
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 application(s). You can start with a lightweight framework 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 bypass much of the monotony inherent in web development, enabling you to get an application up quickly and efficiently.

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 more 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 *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 in 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 I present the main theory, you will type out and then run a small program. I 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. And if you get stuck for more than a few hours, check out the support forum on the Real Python [website](#). 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.

You can download the course files directly from the repository. Press the 'Download ZIP' button which is located at the bottom right of the page. This allows you to download the most recent version of the code as a zip archive. **Be sure to download the updated code for each release.**

Conventions

Formatting

1. Code blocks will be used to present example code.

```
1 print "Hello world"!
```

2. Terminal commands follow the Unix format:

```
1 $ python hello-world.py
```

(dollar signs are not part of the command)

3. *Italic text* will be used to denote a file name:

hello-world.py.

Bold text will be used to denote a new or important term:

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

NOTES, WARNINGS, and SEE ALSO boxes appear as follows:

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

I 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 I 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. 10.8.5, Windows XP, Windows 7, Linux Mint 17, and Ubuntu 14.04 LTS.

Chapter 3

Getting Started

Python Review

Before we begin, you should already have Python installed on your machine. Although Python 3.x has been available since 2008, we'll be using 2.7.6 instead. The majority of web frameworks do not yet support 3.x, because many popular libraries and packages have not been ported from Python version 2 to 3. Fortunately, the differences between 2.7.x and 3.x are minor.

If you do not have Python installed, please refer to Appendix A for a basic tutorial.

To get the most out of this course, I assume you 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.

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 contents of current directory
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:

```
1 $ python big_snake.py
```

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

Windows equivalent:

```
1 C:\> python 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, however the history is erased as soon as you exit the shell.

2. You can use Tab 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, and 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 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 directory (aka, move in one directory).
4. Create a new directory 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:

```
1
2 test
3     test-two
4         test-three
5             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. What directory are you in?
2. What's the fastest way to get to your root directory? HINT: We did not cover this. Use Google. Ask on the support forum.
3. How do you check to see what directories and files are 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 `python` and then pressing enter. The Shell is an interactive environment: You type code directly into it, sending 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:

```
1 $ python
2 >>> phrase = "The bigger the snake, the bigger the prey"
3 >>> print phrase
4 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:

```
1 >>> exit()
2 $
```

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 (about once a day) to buy 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, or 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 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 editor gedit. If you are familiar with a more powerful editor or IDE, feel free to use it. However, I can't stress enough that you must be familiar with it. Do not try to take this course while also learning how to work an IDE. You'll just add another layer of complexity to the entire process.

Really, almost any 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.

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 Python database programming. 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:

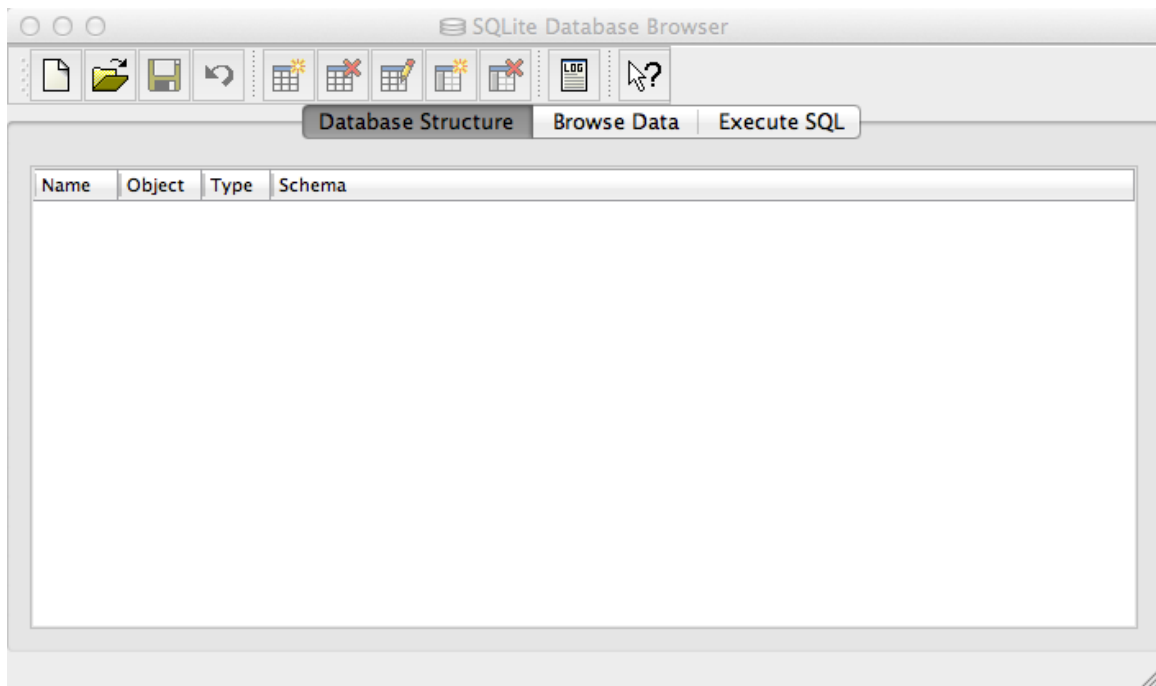


Figure 3.1: SQLite Database Browser

Regardless of the operating system, you can download the SQLite Database Browser from [Sourceforge](#). 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

- Learn the basics of the SQLite Database Browser [here](#).

Installing easy_install and pip

Both easy_install and [pip](#) are Python Package Managers. These essentially make it much easier to install and upgrade Python packages (and package dependencies) by automating the process. In other words, they are 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 it yourself from scratch.

Along with the following instructions, watch the video [here](#) for assistance with installing setup_tools and pip on Windows. Mac Users can follow the instructions [here](#) for installing setup_tools.

easy_install

With easy_install you can simply use the filename of the package you wish to download and install:

```
1 $ easy_install numpy
```

The above command installs numpy or if you already have it installed, it will attempt to install the latest version (if available). If you need a specific version of a package, you can use the following command:

```
1 $ easy_install numpy==1.6.2
```

To delete packages, run the following command and then manually delete the package directory from within your python directory:

```
1 $ easy_install -mxN PackageName numpy
```

To download easy_install, go to the [Python Package Index](#) (PyPI), which is also known as the “[CheeseShop](#)” in the Python community. You need to download setuptools, which includes easy_install. Download the executable (.exe) file for Windows operating systems, or the package egg (.egg) for Unix operating systems (Mac and Linux). You can install it directly from the file. You must have Python installed first before you can install setuptools.

pip

Pip, meanwhile, is similar to easy_install, but there are a few added features:

1. Better error messages

2. Ability to uninstall a package directly from the command line

Also, while `easy_install` will start to install packages before they are done downloading, `pip` waits until downloads are complete. So, if your connection is lost during download, you won't have a partially installed package.

Now, since `pip` is a wrapper that relies on `easy_install`/`Setuptools`, you *must* have `easy_install` setup and working first before you can install `pip`. Once `easy_install` is setup, run the following command to install `pip`:

```
1 $ easy_install pip
```

To install a package:

```
1 $ pip install numpy
```

To download a specific version:

```
1 $ pip install numpy==1.6.1
```

To uninstall a package:

```
1 $ pip uninstall numpy
```

NOTE: If you are in a **Unix environment** you will probably need to use `sudo` before each command in order to execute the [command](#) as the root user: `sudo easy_install pip`. You will then have to enter your root password to install.

Video

Again, please watch the video [here](#) for assistance with installing `setup_tools` and `pip`.

Installing virtualenv

It's common practice to use a [virtualenv](#) (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 virtualenv 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 virtualenv, 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 3.0. Using virtualenv 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, virtualenv allows you to have both versions installed in isolated environments so they don't affect each other.

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

Some Linux flavors come with virtualenv pre-installed. You can run the following command in your terminal to check:

```
1 $ virtualenv --version
```

If installed, the version number will be returned:

```
1 $ virtualenv --version
2 1.8.4
```

Use pip to install virtualenv on your system:

```
1 $ pip install virtualenv
```

Let's practice creating a virtualenv:

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 virtualenv within that directory:

```
1 virtualenv env
```

This created a new directory, “env”, and set up a virtualenv within that directory. The `--no-site-packages` flag truly isolates your work environment from the rest of your system as it does not include any packages or modules already installed on your system. Thus, you have a completely isolated environment, free from any previously installed packages.

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

Unix:

```
1 $ source env/bin/activate
```

Windows:

```
1 $ env\scripts\activate
```

This changes the context of your terminal in that folder, “real-python-test”, which acts as the root of your system. Now you can work with Python in a completely isolated environment. You can tell when you’re working in a virtualenv by the directory surrounded by parentheses to the left of the path in your command-line:

```
1 $ source env/bin/activate
2 (env)Michaels-MacBook-Pro-2:sample michaelherman$
```

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 virtualenv.

Go ahead and deactivate the virtualenv.

Every time you create a new project, install a new virtualenv.

Video

Please watch the video [here](#) for assistance.

Web Browsers

You can either use FireFox or Chrome during this course, because we will make use of a powerful set of tools for web developers called FireBug or Chrome Developer Tools. These tools allow you to inspect and analyze various elements that make up a webpage. You can view HTTP requests, test CSS style changes, and debug code, among others.

If you choose to use FireFox, you will need to install FireBug. I *highly* recommend using Chrome though, as most examples in this course will be shown with Chrome. Plus, Chrome comes with Developer Tools pre-installed.

Install Firebug

Follow these directions if you need to install FireBug.

[Download](#) the latest version of Firefox if you don't already have it. Then follow these steps:

1. On the Menu Bar, click "Tools" => "Add ons"
2. In the "Get Add-ons" tab, search for "firebug".
3. It should be the first search result. Click Install.
4. You may have to restart Firefox for the installation to complete.

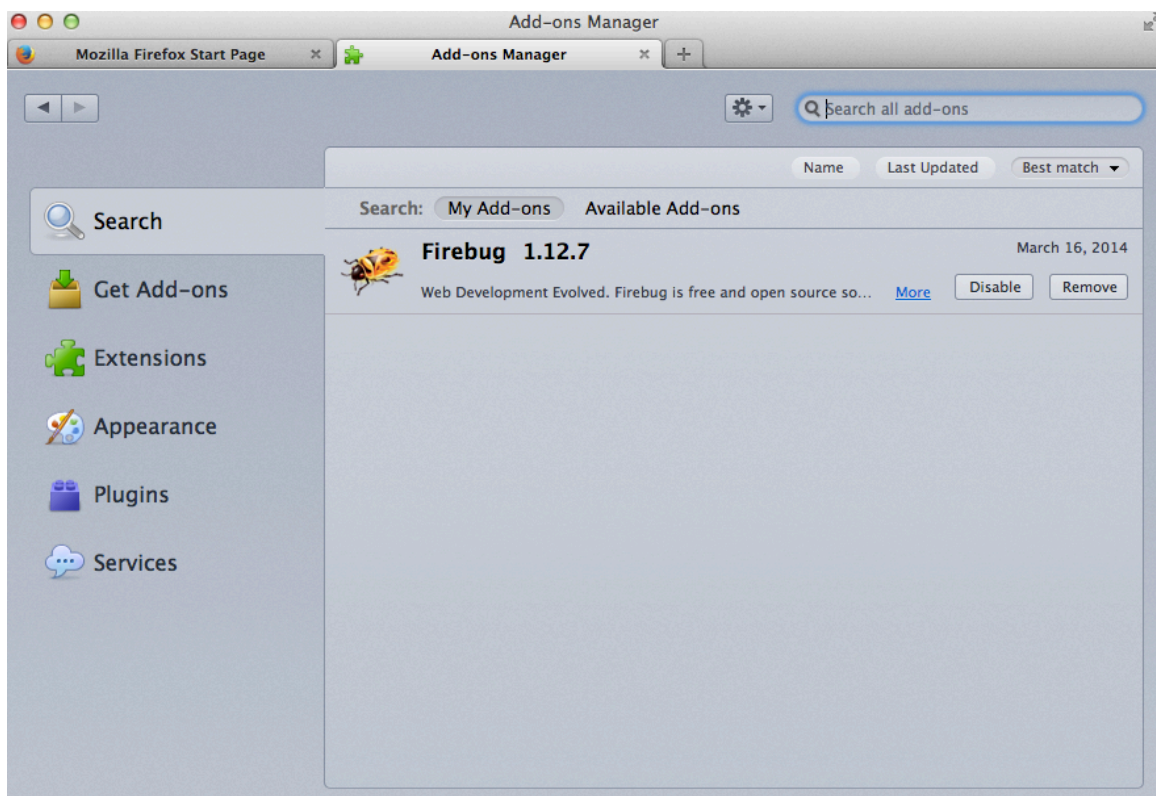


Figure 3.2: Firebug add-on for Firefox

Version Control

Finally, to complete our development environment, 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 comes bigger, it may become necessary to backtrack to an earlier “version” to undo changes if a giant error occurs, for instance. It happens. 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 (like inadvertently deleting a file or a large chunk of code, for example).

Take the time to learn how to use a version control system. This could save you much time in the future - when an error occurs, for example, and you need to rollback your code - and it is a *required* skill for web developers to have.

1. Start by downloading [Git](#), if you don't already have it. Make sure to download the version appropriate for your system.
2. 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:

```
1 $ git config --global user.name "FIRST_NAME LAST_NAME"
2 $ git config --global user.email "MY_NAME@example.com"
```

3. Finally, I 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 overall, high level concepts and work flow. 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 social network used to remotely host those 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 (open source!).

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 setup a new repository on Github. Leave that page up while we create a local repository using Git.

Create a Local Repo

1. Go to your "real-python-test" folder to initialize (create) your local repo (make sure to activate your virtualenv first):

```
1 $ git init
```

2. 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.

```
1 $ touch README.md
```

3. Open that file in Sublime and just type "Hello, world! This is my first PUSH to Github." Save the file.
4. 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:

```
1 $ git add .
```

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 yet part of the local repo yet.

5. Okay. So to add the project to your repo, you need to run the following command:

```
1 $ git commit -am "My initial commit message"
```

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

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

```
1 $ git remote add origin  
https://github.com/<YOUR-USERNAME>/real-python-test.git
```

2. Finally, PUSH (or send) the local repo to Github:

```
1 $ git push origin master
```

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

That’s all there is too it.

Review

Let’s review.

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

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

```
1 $ git init  
2 $ touch README.md  
3 $ git add .  
4 $ git commit -am "message"  
5 $ git remote add origin  
https://github.com/<YOUR-USERNAME>/<YOUR-REPO-NAME>.git  
6 $ git push origin master
```

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

3. 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:

```
1 $ git add .
2 $ git commit -am 'message'
3 $ git push origin master
```

NOTE: The string within the commit command should be replaced each time with a brief description of the changes made since the last PUSH.

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

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

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

Git Workflow

Now that you know how to setup a repo, let's review. So, 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:

```
1 $ git add .
2 $ git commit -am "some message goes here about the commit"
3 $ 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. So, we have one directory, “RealPython”, that will contain all of your projects.
2. Each of those project directories will contain a separate virtualenv, Git repo, and *.gitignore* file.
3. Finally, we want to setup a *requirements.txt* file for *each* virtualenv. 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 virtualenv is activated:

```
1 $ pip freeze > requirements.txt
```

Again, you have your main directory called “RealPython” then within that directory, you’ll create several project directories, each containing - a virtualenv, a Git repo, a *.gitignore* file, and a *requirements.txt* file.

Note: Although this is optional, I 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](#). It’s easy.

Homework

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

Chapter 4

Interlude: Modern Web Development

Overview

Even though this is a Python course, we will be using a number of technologies such as HTML/CSS, JavaScript/jQuery, AJAX, REST, and SQL, to name a few. Put another way, you will be learning all the technologies needed, from the front-end to the back-end (and everything in between), which work in conjunction with Python web frameworks.

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 provides the structure, while CSS provides a pretty facade, and JavaScript/jQuery creates interaction. Essentially, this is what the end user sees when s/he interacts 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 (more on this later).
2. **Middleware:** This layer relays information between the Front and Back-ends, 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 and often times 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.

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/jQuery skills, have focused on the front-end. These roles are becoming less and less defined, however - 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 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, for example, 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](#).

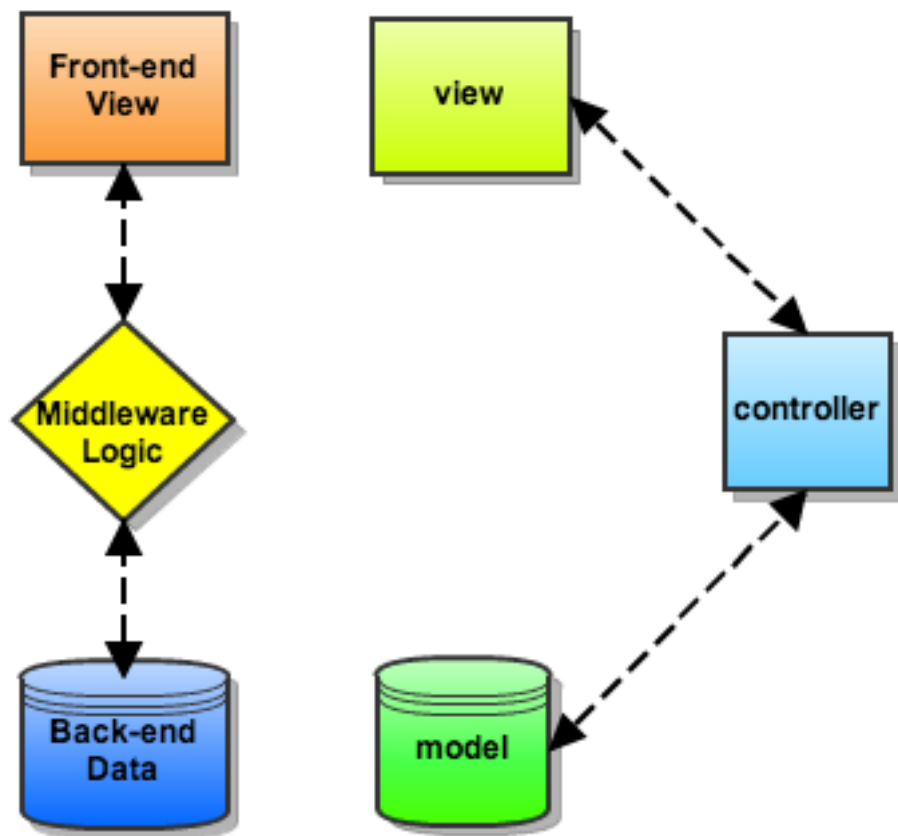


Figure 4.1: Model-View-Controller pattern

Chapter 5

Flask: Quick Start

This chapter introduces you to the Flask web framework.



Figure 5.1: Flask Logo

Overview

[Flask](#) grew from an elaborate [April fool's joke](#) that mocked single file, micro frameworks (most notably [Bottle](#)) 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. Let's take a look.

Installation

1. Within your “RealPython” directory create a new directory called “flask-hello-world”.
2. Navigate into the directory, create and activate a new virtualenv.
3. Now install Flask:

```
1 $ 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!

1. Open Sublime and within a new file add the following code:

```
1 # ---- Flask Hello World ---- #
2
3 # import the Flask class from the flask package
4 from flask import Flask
5
6 # create the application object
7 app = Flask(__name__)
8
9 # use decorators to link the function to a url
10 @app.route("/")
11 @app.route("/hello")
12
13 # define the view using a function, which returns a string
14 def hello_world():
15     return "Hello, World!"
16
17 # start the development server using the run() method
18 if __name__ == "__main__":
19     app.run()
```

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

```
1 $ python app.py
```

3. This creates a test server (or development server), 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 locally, you can point your browser to either URL. Both will work.

4. Test out the URL <http://127.0.0.1:5000/hello> as well.

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 function:

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 if __name__ == "__main__":
6     app.run()
```

1. First, we imported the Flask class from the flask library in order to create our web application.
2. Next, we created an instance of the Flask class and assigned it to the variable app.
3. Finally, we used the `run()` method to run the app locally. By setting the `__name__` variable equal to `"__main__"`, we indicated that we're running the statements in the current file (as a module) rather than importing it. Does this make sense? If not, check out [this](#) link. You can read more about [modules](#) from the Python official documentation.

Now for the functions:

```
1 # use decorators to link the function to a url
2 @app.route("/")
3 @app.route("/hello")
4
5 # define the view using a function, which returns a string
6 def hello_world():
7     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.

```
1 $ git add .
2 $ git commit -am "flask-hello-world"
3 $ 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!

Flow of the App

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

1. The end user (you) requests to view a web page at the URL <http://127.0.0.1:5000/hello>.
2. The controller handles the request determining what should be displayed back, based on the URL that was entered as well as the requested HTTP method (more on this later):

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

3. Since the end user (again, you) just wants to see the text “Hello, World!”, the controller can just render the HTML. 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.
4. The HTML is rendered and displayed to the end user:

```
1 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 quickly look at a dynamic route.

1. Add a new route to *app.py*:

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

Test it out.

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

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

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

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

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

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
- `<int:value>` is treated as an integer
- `<float:value>` is treated as a floating point
- `<path/of/some/sort>` is treated as a path

Test this out. Create three new views:

```

1 @app.route("/integer/<int:value>")
2 def int_type(value):
3     print value + 1
4     return "correct"
5
6 @app.route("/float/<float:value>")
7 def float_type(value):
8     print value + 1
9     return "correct"
10
11 # dynamic route that accepts slashes
12 @app.route("/path/<path:value>")
13 def path_type(value):
14     print value
15     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:

```
1 (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 “text/html” 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.” Then, click the “Network” tab within Developer Tools.

Refresh your page.

Watch Developer Tools:

hi

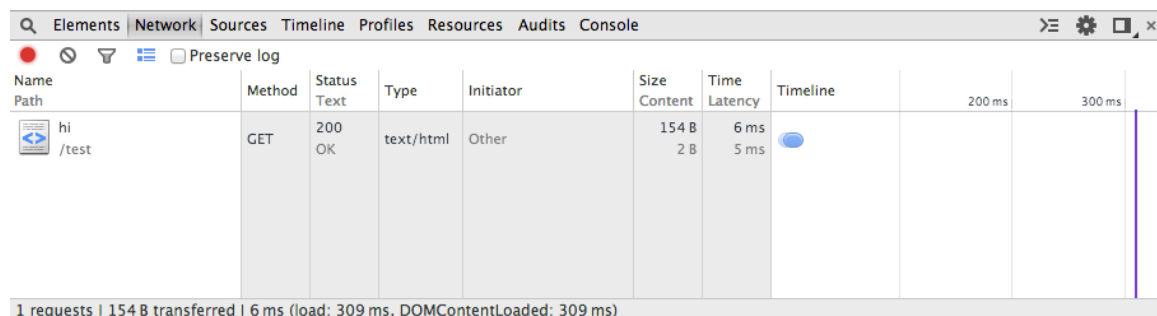


Figure 5.2: Flask Response Information within Chrome Developer Tools

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

Let's add a new view:

```
1 @app.route("/name/<name>")
2 def index(name):
3     if name.lower() == "michael" :
```

```
4     return "Hello, {}".format(name), 200
5 else :
6     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 are also explicitly assigning 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:

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

Same response as before, right? 200 Status Code and a Content-Type of “text/html”. Flask is smart: it inferred the correct Status Code and Content-Type based on the response type. (Some call this “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.

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 snippet to your *app.py* file:

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

The code should now look like this:

```
1 # import the Flask class from the flask module
2 from flask import Flask
3
4 # create the application object
5 app = Flask(__name__)
6
7 # error handling
8 app.config["DEBUG"] = True
9
10 # use decorators to link the function to a url
11 @app.route("/")
12 @app.route("/hello")
13
14 # define the view using a function, which returns a string
15 def hello_world():
16     return "Hello, World!"
17
18 # dynamic route
19 @app.route("/test/<search_query>")
20 def search(search_query):
21     return search_query
22
23 # dynamic route with explicit status codes
24 @app.route("/name/<name>")
25 def index(name):
26     if name.lower() == "michael" :
27         return "Hello, {}".format(name)
28     else :
29         return "Not Found", 404
30
31 # start the development server using the run() method
```

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

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

```
1 * Restarting with reloader
```

Essentially, any time you make changes to the code and save them, 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:

```
1 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 virtualenv.

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

Chapter 6

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. If you think about it, 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 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, UPDATE, INSERT, and DELETE.

Command	Action
SELECT	retrieves data from the database
UPDATE	updates data from the database
INSERT	inserts data into 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 (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. 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 Real Python [course](#), go through chapter 13 again.

Creating Tables

Let's begin. Make sure you've created a "sql" directory (again within your "RealPython" directory). Then create and activate your virtualenv as well as a Git repo.

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

```
1 create table table_name
2 (column1 data_type,
3  column2 data_type,
4  column3 data_type);
```

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

```
1 # Create a SQLite3 database and table
2
3
4 # import the sqlite3 library
5 import sqlite3
6
7 # create a new database if the database doesn't already exist
8 conn = sqlite3.connect("new.db")
9
10 # get a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 # create a table
14 cursor.execute("""CREATE TABLE population
15                 (city TEXT, state TEXT, population INT)
16                 """)
17
18 # close the database connection
19 conn.close()
```

Save the file as *sqla.py*. Run the file from your terminal:

```
1 $ python sqla.py
```

As long as you didn't receive an error, 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):

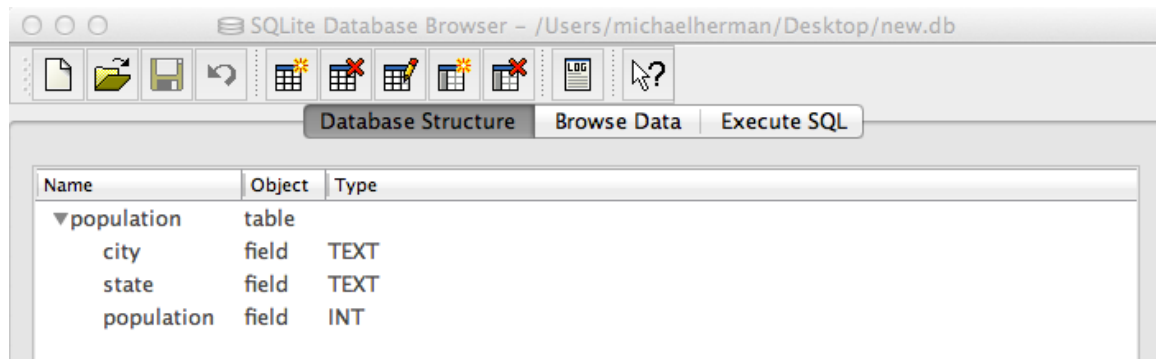


Figure 6.1: SQLite Database Browser

So what exactly did we do here?

1. We imported the `sqlite3` library to communicate with SQLite.
2. Next, we created a new database named *new.db*. (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:

```
1 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 the screen or conduct analysis (SELECT). Go back and look at the SQL statements, from the beginning of the chapter, for a basic review.

You can delete a table by using the `DROP TABLE` command plus the table name you'd like to drop: `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!

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.

```
1 # INSERT Command
2
3
4 # import the sqlite3 library
5 import sqlite3
6
7 # create the connection object
8 conn = sqlite3.connect("new.db")
9
10 # get a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 # insert data
14 cursor.execute("INSERT INTO population VALUES('New York City', \
15             'NY', 8200000)")
16 cursor.execute("INSERT INTO population VALUES('San Francisco', \
17             'CA', 800000)")
18
19 # commit the changes
20 conn.commit()
21
22 # close the database connection
23 conn.close()
```

Save the file as *sqlb.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 rather than double 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:

```
1 import sqlite3
2 with sqlite3.connect("new.db") as connection:
3     c = connection.cursor()
4     c.execute("INSERT INTO population VALUES('New York City', \
5         'NY', 8200000)")
6     c.execute("INSERT INTO population VALUES('San Francisco', \
7         'CA', 800000)")
```

Using the `executemany()` method

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

```
1 # executemany() method
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     # insert multiple records using a tuple
10    cities = [
11        ('Boston', 'MA', 600000),
```

```

12         ('Chicago', 'IL', 2700000),
13         ('Houston', 'TX', 2100000),
14         ('Phoenix', 'AZ', 1500000)
15     ]
16
17     # insert data into table
18     c.executemany('INSERT INTO population VALUES(?, ?, ?)', cities)

```

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

In this example, the question marks (?) act as placeholders (called parameterized statements) for the tuple instead of string substitution (%s). Parameterized 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.

Use the *employees.csv* file for this exercise.

```

1  # import from CSV
2
3  # import the csv library
4  import csv
5
6  import sqlite3
7
8  with sqlite3.connect("new.db") as connection:
9      c = connection.cursor()
10
11     # open the csv file and assign it to a variable
12     employees = csv.reader(open("employees.csv", "rU"))

```

```

13
14     # create a new table called employees
15     c.execute("CREATE TABLE employees(firstname TEXT, lastname
16               TEXT)")
17
18     # insert data into table
19     c.executemany("INSERT INTO employees(firstname, lastname)
20                   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, “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:

```

1  # INSERT Command with Error Handler
2
3
4  # import the sqlite3 library
5  import sqlite3
6
7  # create the connection object
8  conn = sqlite3.connect("new.db")
9
10 # get a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 try:
14     # insert data
15     cursor.execute("INSERT INTO populations VALUES('New York City',
16               'NY', 8200000)")
17     cursor.execute("INSERT INTO populations VALUES('San Francisco',
18               'CA', 800000)")
19
20     # commit the changes
21     conn.commit()
22 except sqlite3.OperationalError:

```

```
21     print "Oops!  Something went wrong. Try again..."
22
23 # close the database connection
24 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:

```
1 # SELECT statement
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     # use a for loop to iterate through the database, printing the
10    # results line by line
11    for row in c.execute("SELECT firstname, lastname from
12                           employees"):
13        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 output the data with just the values by removing the unicode characters altogether:

```
1 # SELECT statement, remove unicode characters
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     c.execute("SELECT firstname, lastname from employees")
10
11    # fetchall() retrieves all records from the query
12    rows = c.fetchall()
13
14    # output the rows to the screen, row by row
15    for r in rows:
16        print r[0], r[1]
```

1. First, the `fetchall()` method retrieved all records from the query and stored them as a tuple - or, more precisely: 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.

```
1 # UPDATE and DELETE statements
2
3 import sqlite3
4
5 with sqlite3.connect("new.db") as connection:
6     c = connection.cursor()
7
8     # update data
9     c.execute("UPDATE population SET population = 9000000 WHERE
10               city='New York City'")
11
12     # delete data
13     c.execute("DELETE FROM population WHERE city='Boston'")
14
15     print "\nNEW DATA:\n"
16
17     c.execute("SELECT * FROM population")
18
19     rows = c.fetchall()
20
21     for r in rows:
22         print r[0], r[1], r[2]
```

1. In this example, 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:

```
1 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. Please 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.
- Update the quantity on two of the records, and then output all of the records from the table.
- Finally output only records that are for Ford vehicles.

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.

```
1 # executemany() method
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     # insert multiple records using a tuple
10    # (you can copy and paste the values)
11    cities = [
12        ('Boston', 'MA', 600000),
13        ('Los Angeles', 'CA', 38000000),
14        ('Houston', 'TX', 2100000),
15        ('Philadelphia', 'PA', 1500000),
16        ('San Antonio', 'TX', 1400000),
17        ('San Diego', 'CA', 130000),
18        ('Dallas', 'TX', 1200000),
19        ('San Jose', 'CA', 900000),
20        ('Jacksonville', 'FL', 800000),
21        ('Indianapolis', 'IN', 800000),
22        ('Austin', 'TX', 800000),
23        ('Detroit', 'MI', 700000)
24    ]
25
26    c.executemany("INSERT INTO population VALUES(?, ?, ?)", cities)
27
28    c.execute("SELECT * FROM population WHERE population > 1000000")
29
30    rows = c.fetchall()
31
32    for r in rows:
33        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:

```
1 # Create a table and populate it with data
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     c.execute("""CREATE TABLE regions
10                (city TEXT, region TEXT)
11                """)
12
13 # (again, copy and paste the values if you'd like)
14 cities = [
15     ('New York City', 'Northeast'),
16     ('San Francisco', 'West'),
17     ('Chicago', 'Midwest'),
18     ('Houston', 'South'),
19     ('Phoenix', 'West'),
20     ('Boston', 'Northeast'),
21     ('Los Angeles', 'West'),
22     ('Houston', 'South'),
23     ('Philadelphia', 'Northeast'),
24     ('San Antonio', 'South'),
25     ('San Diego', 'West'),
26     ('Dallas', 'South'),
27     ('San Jose', 'West'),
28     ('Jacksonville', 'South'),
29     ('Indianapolis', 'Midwest'),
30     ('Austin', 'South'),
31     ('Detroit', 'Midwest')
32 ]
33
34 c.executemany("INSERT INTO regions VALUES(?, ? )", cities)
35
```

```

36     c.execute("SELECT * FROM regions ORDER BY region ASC")
37
38     rows = c.fetchall()
39
40     for r in rows:
41         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:

```

1  # JOINING data from multiple tables
2
3
4  import sqlite3
5
6  with sqlite3.connect("new.db") as connection:
7      c = connection.cursor()
8
9      # retrieve data
10     c.execute("""SELECT population.city, population.population,
11                 regions.region FROM population, regions
12                 WHERE population.city = regions.city""")
13
14     rows = c.fetchall()
15
16     for r in rows:
17         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:

```
1 # JOINing data from multiple tables - cleanup
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     c.execute("""SELECT DISTINCT population.city,
10                  population.population,
11                  regions.region FROM population, regions WHERE
12                  population.city = regions.city ORDER by
13                  population.city ASC""")
14
15 rows = c.fetchall()
16
17 for r in rows:
18     print "City: " + r[0]
19     print "Population: " + str(r[1])
20     print "Region: " + r[2]
21     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).
- 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.

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

```
1 # SQLite Functions
2
3 import sqlite3
4
5 with sqlite3.connect("new.db") as connection:
6     c = connection.cursor()
7
8     # create a dictionary of sql queries
9     sql = {'average': "SELECT avg(population) FROM population",
10           'maximum': "SELECT max(population) FROM population",
11           'minimum': "SELECT min(population) FROM population",
12           'sum': "SELECT sum(population) FROM population",
13           'count': "SELECT count(city) FROM population"}
14
15     # run each sql query item in the dictionary
16     for keys, values in sql.items():
17
18         # run sql
19         c.execute(values)
20
21         # fetchone() retrieves one record from the query
22         result = c.fetchone()
23
24         # output the result to screen
25         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.
- 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!**

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 bites that you can handle. You've already learned the material; we're just putting it all together. Spend some time mapping (drawing) out the workflow as a first step.

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 he or she would like to perform an aggregation (AVG, MAX, MIN, or SUM) or exit the program altogether.

Break this assignment into two scripts. Name them *assignment3a.py* and *assignment3b.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

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

```
1 import sqlite3
2 import random
```

2. Establish connection and create *newnum.db* database:

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

3. Open the cursor:

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

4. Create table, "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):


```

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

```

5. Use a for loop and random function to insert 100 random values (0 to 100):

```

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

```

Full Code:

```

1 # Assignment 3a - insert random data
2
3
4 # import the sqlite3 library
5 import sqlite3
6 import random
7
8 with sqlite3.connect("newnum.db") as connection:
9     c = connection.cursor()
10
11     # delete database table if exist
12     c.execute("DROP TABLE if exists numbers")
13
14     # create database table
15     c.execute("CREATE TABLE numbers(num int)")
16
17     # insert each number to the database
18     for i in range(100):
19         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, then 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, 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:

```
1 # Assignment 3b - prompt the user
2
3
4 # import the sqlite3 library
5 import sqlite3
6
7 # create the connection object
8 conn = sqlite3.connect("newnum.db")
9
10 # create a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 prompt = """
14 Select the operation that you want to perform [1-5]:
15 1. Average
16 2. Max
17 3. Min
18 4. Sum
19 5. Exit
20 """
21
22 # loop until user enters a valid operation number [1-5]
23 while True:
24     # get user input
25     x = raw_input(prompt)
26
27     # if user enters any choice from 1-4
28     if x in set(["1", "2", "3", "4"]):
```

```

29     # parse the corresponding operation text
30     operation = {1: "avg", 2:"max", 3:"min", 4:"sum"}[int(x)]
31
32     # retrieve data
33     cursor.execute("SELECT {}(num) from
34                     numbers".format(operation))
35
36     # fetchone() retrieves one record from the query
37     get = cursor.fetchone()
38
39     # output result to screen
40     print operation + ":  %f" % get[0]
41
42     # if user enters 5
43     elif x == "5":
44         print "Exit"
45
46     # exit loop
47     break

```

We asked the user to enter the operation they would like to perform (numbers 1-4), 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 ...

1. Inserting:

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

2. Updating:

```
1 UPDATE table_name
2 SET column1=value1
3 WHERE some_column=some_value;
```

3. Deleting

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

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 are not 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?

Chapter 7

Flask Blog App

Let's build a blog!

Requirements:

1. After a user logs in he or she is 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 logout.

That's it.

Project Structure

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

```
1
2  blog.py
3  static
4      css
5      img
6      js
7  templates
```

First, all of the logic (Python/Flask code) goes in the *blog.py* file. Our “static” directory holds static files 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 (i.e., the web browser), then returns content back to the client in the form of a response:

NOTE: HTTP is the method used for all web-based communications; the ‘http://’ that prefixes URLs designates an HTTP request. Literally everything you see in your browser is transferred to you via HTTP.

5. Install flask:

```
1 pip install flask
```

Let’s build our app!

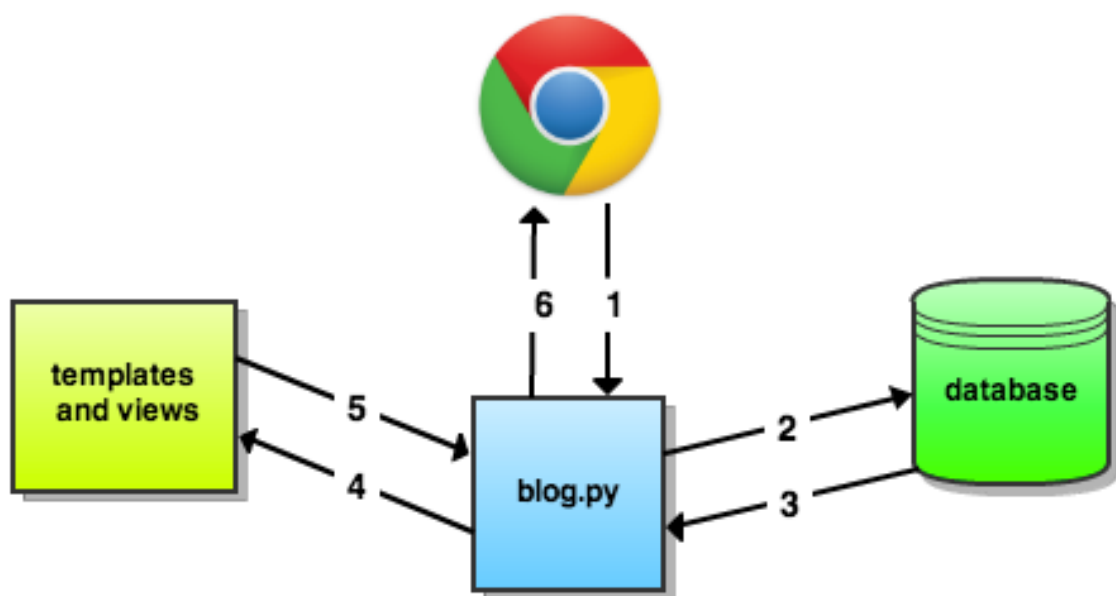


Figure 7.1: HTTP request/response Cycle

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:

```
1 # sql.py - Create a SQLite3 table and populate it with data
2
3
4 import sqlite3
5
6 # create a new database if the database doesn't already exist
7 with sqlite3.connect("blog.db") as connection:
8
9     # get a cursor object used to execute SQL commands
10    c = connection.cursor()
11
12    # create the table
13    c.execute("""CREATE TABLE posts
14                (title TEXT, post TEXT)
15                """)
16
17    # insert dummy data into the table
18    c.execute('INSERT INTO posts VALUES("Good", "I\'m good.")')
19    c.execute('INSERT INTO posts VALUES("Well", "I\'m well.")')
20    c.execute('INSERT INTO posts VALUES("Excellent", "I\'m
21                excellent.")')
22    c.execute('INSERT INTO posts VALUES("Okay", "I\'m okay.")')
```

Save the file within your “flask-blog” directory. Run it. This reads the database definition provided in *sql.py* and then creates the actual database file 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 `hello_world` app (*app.py*), this script will define the imports, configurations, and each view.

```
1 # blog.py - controller
2
3
4 # imports
5 from flask import Flask, render_template, request, session, \
6     flash, redirect, url_for, g
7 import sqlite3
8
9 # configuration
10 DATABASE = 'blog.db'
11
12 app = Flask(__name__)
13
14 # pulls in app configuration by looking for UPPERCASE variables
15 app.config.from_object(__name__)
16
17 # function used for connecting to the database
18 def connect_db():
19     return sqlite3.connect(app.config['DATABASE'])
20
21 if __name__ == '__main__':
22     app.run(debug=True)
```

Save this file as *blog.py* in your main project directory. The `configuration` section is used for defining application-specific settings.

Make sure you understand how this is working:

```
1 # configuration
2 DATABASE = 'blog.db'
3
4 ...snip...
5
6 # pulls in app configuration by looking for UPPERCASE variables
7 app.config.from_object(__name__)
```

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

Views

After a user logs in, s/he is 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.

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

We also need a login page:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <h3>Please login to access your blog.</h3>
5     <p>Temp Login: <a href="/main">Login</a></p>
6 {% 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:

```
1 @app.route('/')
2 def login():
3     return render_template('login.html')
4
5 @app.route('/main')
6 def main():
7     return render_template('main.html')
```

Updated code:

```
1 # blog.py - controller
2
3
4 # imports
5 from flask import Flask, render_template, request, session, \
6     flash, redirect, url_for, g
7 import sqlite3
```

```

8
9 # configuration
10 DATABASE = 'blog.db'
11
12 app = Flask(__name__)
13
14 # pulls in configurations by looking for UPPERCASE variables
15 app.config.from_object(__name__)
16
17 # function used for connecting to the database
18 def connect_db():
19     return sqlite3.connect(app.config['DATABASE'])
20
21 @app.route('/')
22 def login():
23     return render_template('login.html')
24
25 @app.route('/main')
26 def main():
27     return render_template('main.html')
28
29 if __name__ == '__main__':
30     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. Say it out loud.

Templates

Templates (*template.html*, in our case) are HTML skeletons that serve as the base for either your entire website or pieces of your website. 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:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Welcome, friends!</title>
5   </head>
6   <body>
7     <div class="container">
8       {% block content %}
9       {% endblock %}
10    </div>
11  </body>
12 </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, our 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*.

You may have also noticed 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:

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.

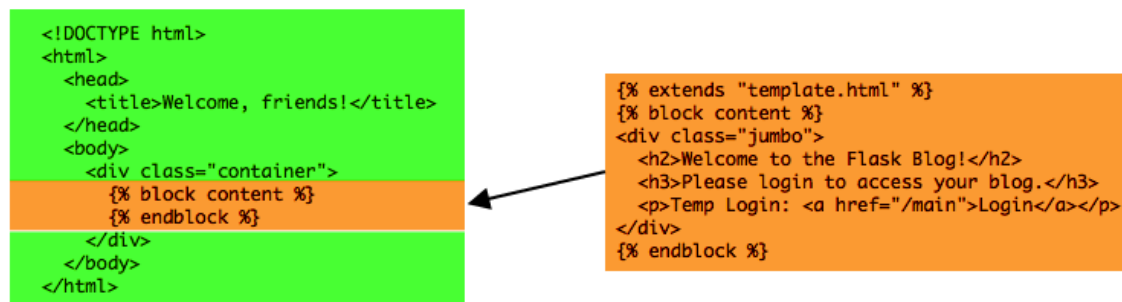


Figure 7.2: Parent and Child Jinja2 templates

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.

Run the server!

All right! 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 mine.

What's going on?

So, 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 code 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 login to so that they can access the main blog page.

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

```
1 USERNAME = 'admin'
2 PASSWORD = 'admin'
```

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

```
1 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:

```
1 >>> import os
2 >>> os.urandom(24)
3 '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:

```
1 # configuration
2 DATABASE = 'blog.db'
3 USERNAME = 'admin'
4 PASSWORD = 'admin'
5 SECRET_KEY = 'hard_to_guess'
```

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

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     if request.method == 'POST':
5         if request.form['username'] != app.config['USERNAME'] or \
6             request.form['password'] != app.config['PASSWORD']:
```

```

7         error = 'Invalid Credentials. Please try again.'
8     else:
9         session['logged_in'] = True
10        return redirect(url_for('main'))
11    return render_template('login.html', error=error)

```

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.

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

For example:

```

1 >>> from flask import Flask, url_for
2 >>> app = Flask(__name__)
3 >>> @app.route('/')
4 ... def index():
5 ...     pass
6 ...
7 >>> @app.route('/main')
8 ... def main():
9 ...     pass
10 ...
11 >>> with app.test_request_context():
12 ...     print url_for('index')
13 ...     print url_for('main')
14 ...
15 /
16 /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 *aloud* what each line accomplishes.

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

```

1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <h3>Please login to access your blog.</h3>
5     <form action="" method="post">
6         Username: <input type="text" name="username" value="{{
7             request.form.username }}">
8         Password: <input type="password" name="password" value="{{
9             request.form.password }}">
10        <p><input type="submit" value="Login"></p>
11    </form>
12 {% endblock %}

```

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

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

```

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

```

The `logout()` function uses the `pop()` method 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.

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

```

1 {% for message in get_flashed_messages() %}
2     <div class="flash">{{ message }}</div>
3 {% endfor %}
4 {% if error %}
5     <p class="error"><strong>Error:</strong> {{ error }}</p>
6 {% endif %}

```

-so that the template now looks like this:

```

1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Welcome, friends!</title>

```

```

5 </head>
6 <body>
7   <div class="container">
8     {% for message in get_flashed_messages() %}
9       <div class="flash">{{ message }}</div>
10    {% endfor %}
11    {% if error %}
12      <p class="error"><strong>Error:</strong> {{ error }}</p>
13    {% endif %}
14    <!-- inheritance -->
15    {% block content %}
16    {% endblock %}
17    <!-- end inheritance -->
18  </div>
19 </body>
20 </html>

```

Finally, add a logout link to the *main.html* page:

```

1 {% extends "template.html" %}
2 {% block content %}
3   <h2>Welcome to the Flask Blog!</h2>
4   <p><a href="/logout">Logout</a></p>
5 {% endblock %}

```

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

Sessions and Login_required Decorator

Now that users are able to login and logout, 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? This is not good.

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 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 (e.g., `logged_in`) 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*:

```
1 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, setup the new function in *blog.py*:

```
1 def login_required(test):
2     @wraps(test)
3     def wrap(*args, **kwargs):
4         if 'logged_in' in session:
5             return test(*args, **kwargs)
6         else:
7             flash('You need to login first.')
8             return redirect(url_for('login'))
9     return wrap
```

This tests 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 login is required.

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

```
1 @app.route('/main')
2 @login_required
3 def main():
```

```
4     return render_template('main.html')
```

Updated code:

```
1 # blog.py - controller
2
3
4 # imports
5 from flask import Flask, render_template, request, session, \
6     flash, redirect, url_for, g
7 import sqlite3
8 from functools import wraps
9
10 # configuration
11 DATABASE = 'blog.db'
12 USERNAME = 'admin'
13 PASSWORD = 'admin'
14 SECRET_KEY = 'hard_to_guess'
15
16 app = Flask(__name__)
17
18 # pulls in configurations by looking for UPPERCASE variables
19 app.config.from_object(__name__)
20
21 # function used for connecting to the database
22 def connect_db():
23     return sqlite3.connect(app.config['DATABASE'])
24
25 def login_required(test):
26     @wraps(test)
27     def wrap(*args, **kwargs):
28         if 'logged_in' in session:
29             return test(*args, **kwargs)
30         else:
31             flash('You need to login first.')
32             return redirect(url_for('login'))
33     return wrap
34
35 @app.route('/', methods=['GET', 'POST'])
36 def login():
37     error = None
```

```

38     if request.method == 'POST':
39         if request.form['username'] != app.config['USERNAME'] or \
40             request.form['password'] != app.config['PASSWORD']:
41             error = 'Invalid Credentials. Please try again.'
42         else:
43             session['logged_in'] = True
44             return redirect(url_for('main'))
45     return render_template('login.html', error=error)
46
47 @app.route('/main')
48 @login_required
49 def main():
50     return render_template('main.html')
51
52 @app.route('/logout')
53 def logout():
54     session.pop('logged_in', None)
55     flash('You were logged out')
56     return redirect(url_for('login'))
57
58 if __name__ == '__main__':
59     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, he or she is redirected back to the login screen.

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

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. Login successful:

```

1 127.0.0.1 - - [01/Feb/2014 11:37:56] "POST / HTTP/1.1" 302 -
2 127.0.0.1 - - [01/Feb/2014 11:37:56] "GET /main HTTP/1.1" 200 -

```

Here, the login credentials were sent with a 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 on the client side within a cookie. You can view this token in your browser by opening up Developer Tools in Chrome, clicking the “Resources” tab, then looking at your cookies:

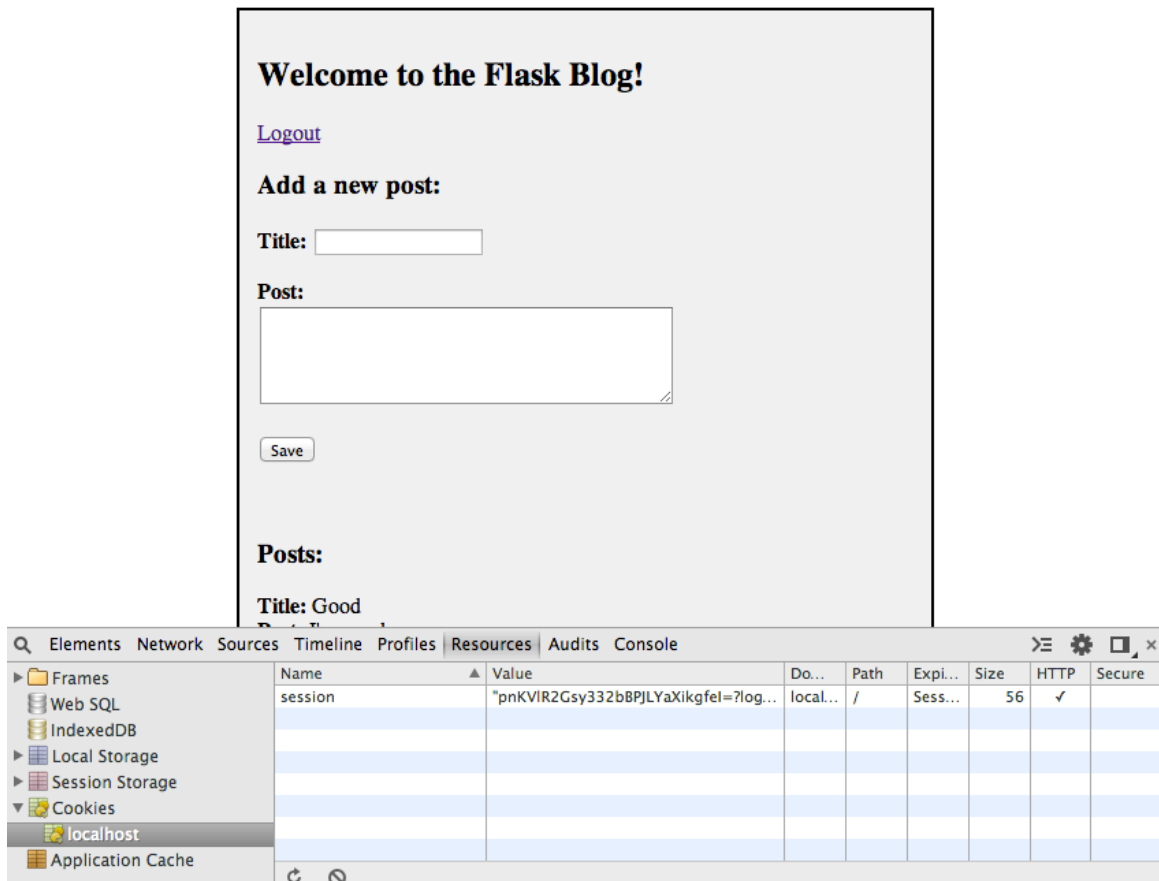


Figure 7.3: Cookies in Chrome Developer Tools

2. Logout:

```

1 127.0.0.1 - - [01/Feb/2014 11:38:53] "GET /logout HTTP/1.1" 302 -
2 127.0.0.1 - - [01/Feb/2014 11:38:53] "GET / HTTP/1.1" 200 -

```

When you log out, you are actually issuing a GET request that responds by redirecting you back to *login.html*. Again, this request was successful.

3. Login failed:

```
1 127.0.0.1 - - [01/Feb/2014 11:40:07] "POST / HTTP/1.1" 200 -
```

If you enter the wrong login credentials when trying to login you still get a 200 success code as the server responds with an error.

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

```
1 127.0.0.1 - - [01/Feb/2014 11:44:56] "GET /main HTTP/1.1" 302 -
2 127.0.0.1 - - [01/Feb/2014 11:44:56] "GET / HTTP/1.1" 200 -
```

If you try to access *main.html* without logging in first, you will be redirected back to *login.html*.

The server log/stack trace comes in handy when you need to debug your code. Let's say, for example, that 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:

```
1 127.0.0.1 - - [01/Feb/2014 11:52:31] "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 with respect to debugging errors.

Show Posts

Now that basic security is set up, we need to display some information to the user. Otherwise, what's the point of the user logging in in the first place? Let's start by displaying the current posts. Update the `main()` function within *blog.py*:

```
1 @app.route('/main')
2 @login_required
3 def main():
4     g.db = connect_db()
5     cur = g.db.execute('select * from posts')
6     posts = [dict(title=row[0], post=row[1]) for row in
7               cur.fetchall()]
8     g.db.close()
9     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()]` assigns the data retrieved from the database to a dictionary, which is assigned to the variable `posts`.
4. `posts=posts` passes that variable to the *main.html* file.

We next need to edit *main.html* to loop through the dictionary in order to display the titles and posts:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <p><a href="/logout">Logout</a></p>
5     <br/>
6     <br/>
7     <h3>Posts:</h3>
8     {% for p in posts %}
9         <strong>Title:</strong> {{ p.title }} <br/>
```

```
10     <strong>Post:</strong> {{ p.post }} <br/>
11     <br/>
12     {% endfor %}
13 {% 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 variable 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()`:

```
1 @app.route('/add', methods=['POST'])
2 @login_required
3 def add():
4     title = request.form['title']
5     post = request.form['post']
6     if not title or not post:
7         flash("All fields are required. Please try again.")
8         return redirect(url_for('main'))
9     else:
10        g.db = connect_db()
11        g.db.execute('insert into posts (title, post) values (?,
12                        ?)',
13                        [request.form['title'], request.form['post']])
14        g.db.commit()
15        g.db.close()
16        flash('New entry was successfully posted!')
17        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 table.

NOTE: The above description is a high-level overview of what's really happening. Get granular with it. Read what each line accomplishes aloud. Repeat to a friend.

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

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <p><a href="/logout">Logout</a></p>
5     <h3>Add a new post:</h3>
6     <form action="{% url_for('add') %}" method="post" class="add">
7         <label><strong>Title:</strong></label>
8         <input name="title" type="text">
9         <p><label><strong>Post:</strong></label><br>
```

```

10     <textarea name="post" rows="5" cols="40"></textarea></p>
11     <input class="button" type="submit" value="Save">
12 </form>
13 <br>
14 <br>
15 <h3>Posts:</h3>
16 {% for p in posts %}
17     <strong>Title:</strong> {{ p.title }} <br>
18     <strong>Post:</strong> {{ p.post }} <br>
19     <br>
20 {% endfor %}
21 {% endblock %}

```

Test this out! Make sure to add a post.

We issued an HTTP POST request to submit the form to the `add()` function, which then redirected us back to `main.html` with the new post:

```

1 127.0.0.1 - - [01/Feb/2014 12:14:24] "POST /add HTTP/1.1" 302 -
2 127.0.0.1 - - [01/Feb/2014 12:14:24] "GET /main HTTP/1.1" 200 -

```

Style

All right. 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. For now, please just follow along.*

```
1 .container {
2   background: #f4f4f4;
3   margin: 2em auto;
4   padding: 0.8em;
5   width: 30em;
6   border: 2px solid #000;
7 }
8
9 .flash, .error {
10  background: #000;
11  color: #fff;
12  padding: 0.5em;
13 }
```

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

```
1 <link rel="stylesheet"
2     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. In other words, this translates to: “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 me the CSS, so I can make mine look better.

Welcome to the Flask Blog!

Please login to access your blog.

Username: Password:

Figure 7.4: Flask Blog with updated styles

Conclusion

Let's recap:

1. First, we used Flask to create a basic website structure to house static pages.
2. Then we added a login 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.

Simple, right?

Make sure to commit to Git and then PUSH to Github!

Homework

- Now take a look at the accompanying [video](#) to see how to deploy your app on PythonAnywhere.

Chapter 8

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` (or “Python DeBugger”) within the standard library, where you can set breakpoints, step through your code, and inspect the stack.

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.

1. Save the following code as *pdb_ex.py* in the “debugging” folder:

```
1 import sys
2 from random import choice
3
4 random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
5 random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
6
7 while True:
8     print "To exit this game type 'exit'"
9     answer = raw_input("What is {} times {}? ".format(
10         choice(random2), choice(random1)))
11
12     # exit
13     if answer == "exit":
14         print "Now exiting game!"
15         sys.exit()
16
17     # determine if number is correct
18     elif answer == choice(random2) * choice(random1):
19         print "Correct!"
20     else:
21         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!

2. Import the `pdb` module:

```
1 import pdb
```

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

```
1 import sys
2 import pdb
3 from random import choice
```

```

4
5 random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
6 random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
7
8 while True:
9     print "To exit this game type 'exit'"
10
11     pdb.set_trace()
12
13     answer = raw_input("What is {} times {}? ".format(
14         choice(random2), choice(random1)))
15
16     # exit
17     if answer == "exit":
18         print "Now exiting game!"
19         sys.exit()
20
21     # determine if number is correct
22     elif answer == choice(random2) * choice(random1):
23         print "Correct!"
24     else:
25         print "Wrong!"

```

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

```

1 $ python pdb_ex.py
2 To exit this game type 'exit'
3 > /debugging/pdb_ex.py(13)<module>()
4 -> answer = raw_input("What is {} times {}? ".format(
5 (Pdb)

```

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

From here you can start stepping through your code to see what happens, line by line. Check out the list of commands you have access to [here](#). There's quite a lot 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.

5. First, see what the value of answer is:

```

1 (Pdb) n
2 > /debugging/pdb_ex.py(14)<module>()
3 -> choice(random2), choice(random1)))
4 (Pdb) n
5 What is 12 times 10? 120
6 > /debugging/pdb_ex.py(17)<module>()
7 -> if answer == "exit":
8 (Pdb) p answer
9 '120'
```

6. Next, let's continue through the program to see if that value (120) changes:

```

1 (Pdb) n
2 > /debugging/pdb_ex.py(22)<module>()
3 -> elif answer == choice(random2) * choice(random1):
4 (Pdb) n
5 > /debugging/pdb_ex.py(25)<module>()
6 -> print "Wrong!"
7 (Pdb) p answer
8 '120'
```

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

7. Update the code for testing:

```

1 import sys
2 from random import choice
```

```

3
4 random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
5 random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
6
7 while True:
8     print "To exit this game type 'exit'"
9     answer = raw_input("What is {} times {}? ".format(
10         choice(random2), choice(random1))
11     )
12
13     # exit
14     if answer == "exit":
15         print "Now exiting game!"
16         sys.exit()
17
18     test = int(choice(random2))*int(choice(random1))
19     # determine if number is correct
20     # elif answer == choice(random2) * choice(random1):
21     #     print "Correct!"
22     # else:
23     #     print "Wrong!"

```

We just took the value we are using to test our answer against and assigned it to a variable, test.

8. Debug time. Exit the debugger, and let's start over.

```

1 $ python pdb_ex.py
2 To exit this game type 'exit'
3 > /debugging/pdb_ex.py(13)<module>()
4 -> answer = raw_input("What is {} times {}? ".format(
5 (Pdb) n
6 > /debugging/pdb_ex.py(14)<module>()
7 -> choice(random2), choice(random1))
8 (Pdb) n
9 What is 2 times 2? 4
10 > /debugging/pdb_ex.py(17)<module>()
11 -> if answer == "exit":
12 (Pdb) n
13 > /debugging/pdb_ex.py(21)<module>()
14 -> test = int(choice(random2))*int(choice(random1))

```

```

15 (Pdb) n
16 > /debugging/pdb_ex.py(8)<module>()
17 -> while True:
18 (Pdb) p test
19 20

```

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

9. Refactor:

```

1 import sys
2 from random import choice
3
4 random1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
5 random2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
6
7 while True:
8     print "To exit this game type 'exit'"
9     num1 = choice(random2)
10    num2 = choice(random1)
11    answer = int(raw_input("What is {} times {}? ".format(num1,
12    num2)))
13
14    # exit
15    if answer == "exit":
16        print "Now exiting game!"
17        sys.exit()
18
19    # determine if number is correct
20    elif answer == num1 * num2:
21        print "Correct!"
22        break
23    else:
24        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 cast the answer to an integer.

10. 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 then add the line number that you wish to break on: `b <line #>`

Simple example:

```
1 import pdb
2
3
4 def add_one(num):
5     result = num + 1
6     print result
7     return result
8
9
10 def main():
11     pdb.set_trace()
12     for num in xrange(0, 10):
13         add_one(num)
14
15
16 if __name__ == "__main__":
17     main()
```

Save this as `pdb_ex2.py`.

Now, let's look at an example:

```
1 python pdb_ex2.py
2 > /debugging/pdb_ex2.py(12)main()
3 -> for num in xrange(0, 10):
4 (Pdb) b 5
5 Breakpoint 1 at /debugging/pdb_ex2.py:5
6 (Pdb) c
7 > /debugging/pdb_ex2.py(5)add_one()
8 -> result = num + 1
9 (Pdb) args
10 num = 0
11 (Pdb) b 13
12 Breakpoint 2 at /debugging/pdb_ex2.py:13
13 (Pdb) c
14 1
15 > /debugging/pdb_ex2.py(13)main()
```

```

16 -> add_one(num)
17 (Pdb) b 5
18 Breakpoint 3 at /debugging/pdb_ex2.py:5
19 (Pdb) c
20 > /debugging/pdb_ex2.py(5)add_one()
21 -> result = num + 1
22 (Pdb) args
23 num = 1
24 (Pdb) c
25 2
26 > /debugging/pdb_ex2.py(13)main()
27 -> 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:

```
1 def add_one_hundred():
2     again = 'yes'
3     while again == 'yes':
4         number = raw_input('Enter a number between 1 and 10: ')
5         new_number = (int(number) + 100)
6         print '{} plus 100 is {}'.format(number, new_number)
7         again = raw_input('Another round, my friend? (`yes` or `no`) ')
8     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?

```
1 >>> from post_mortem_pdb import *
2 >>> add_one_hundred()
3 Enter a number between 1 and 10: test
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "post_mortem_pdb.py", line 5, in add_one_hundred
7     new_number = (int(number) + 100)
8 ValueError: invalid literal for int() with base 10: 'test'
9 >>>
```

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 the PDB to start debugging where the exception occurred:

```
1 >>> import pdb; pdb.pm()
2 > /debugging/post_mortem_pdb.py(5)add_one_hundred()
3 -> new_number = (int(number) + 100)
4 (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 example of just that when we start working with the advanced Flask and Django material later in the course.

Cheers!

Homework

- Watch these two videos on debugging: [one](#) and [two](#).

Chapter 9

Flask: FlaskTaskr, Part 1 - Quick Start

Overview

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

NOTE: This project is Python 3 compatible!

Let's get to it.

For now, this application has the following features:

1. Users sign in and out from the landing page.
2. New users register on a separate registration page.
3. Once signed in, users add new tasks; each task consists of a name, due date, priority, status, and a unique, auto-incremented ID.
4. Users view all uncompleted tasks from the same screen.
5. Users also delete tasks and mark tasks as completed; if a user deletes a task, it will also be deleted from the database.

Before beginning take a moment to review the steps taken 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 repo and PUSH to Github frequently.*

Setup

For simplicity's sake, Since you should be familiar with the workflow, comments and explanations will not address what has already been explained. Please refer to the *Flask Blog App* chapter for a more detailed explanation, as needed.

Let's get started.

1. Navigate to your “realpython” directory, and create a new directory called “flasktaskr”. Navigate into the newly created directory.
2. Create and activate a new virtualenv.
3. Install Flask:

```
1 $ pip install flask==0.10.1
2 $ pip freeze > requirements.txt
```

4. Create a new directory called “project”, which is the project root directory.
5. Setup the following files and directories within the root directory:

```
1
2 static
3     css
4     img
5     js
6 templates
7 views.py
```

6. 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 our blog app's configurations directly in the 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 the main reason is to separate our app's logic from static variables.

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

```
1 import os
2
3 # grab the folder where this script lives
4 basedir = os.path.abspath(os.path.dirname(__file__))
5
6 DATABASE = 'flasktaskr.db'
7 USERNAME = 'admin'
8 PASSWORD = 'admin'
9 WTF_CSRF_ENABLED = True
10 SECRET_KEY = 'my_precious'
11
12 # define the full path for the database
13 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. Always make sure to set the secret key to something that is nearly impossible to guess. Use a [random key generator](#).

Database

Based on the info above (in the Overview section) regarding the main functionality of the app, 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 (boolean) - 1 if the task is open and 0 if closed.

```
1 # project/db_create.py
2
3
4 import sqlite3
5 from _config import DATABASE_PATH
6
7 with sqlite3.connect(DATABASE_PATH) as connection:
8
9     # get a cursor object used to execute SQL commands
10    c = connection.cursor()
11
12    # create the table
13    c.execute("""CREATE TABLE tasks(task_id INTEGER PRIMARY KEY
14                AUTOINCREMENT,
15                name TEXT NOT NULL, due_date TEXT NOT NULL, priority
16                INTEGER NOT NULL,
17                status INTEGER NOT NULL)""")
18
19    # insert dummy data into the table
20    c.execute(
21        'INSERT INTO tasks (name, due_date, priority, status)'
22        'VALUES("Finish this tutorial", "03/25/2015", 10, 1)'
23    )
24    c.execute(
25        'INSERT INTO tasks (name, due_date, priority, status)'
26        'VALUES("Finish Real Python Course 2", "03/25/2015", 10, 1)'
27    )
```

Two things to note:

1. Notice how we did not need to specify the “task_id” when entering (INSERT INTO command) data into the table as it’s an auto-incremented value, which means that it’s auto-generated with each new row of data. 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. Check it out on your own.

Controller

Add the following code to *views.py*:

```
1 # project/views.py
2
3
4 import sqlite3
5 from functools import wraps
6
7 from flask import Flask, flash, redirect, render_template, \
8     request, session, url_for
9
10
11 # config
12
13 app = Flask(__name__)
14 app.config.from_object('_config')
15
16
17 # helper functions
18
19 def connect_db():
20     return sqlite3.connect(app.config['DATABASE_PATH'])
21
22
23 def login_required(test):
24     @wraps(test)
25     def wrap(*args, **kwargs):
26         if 'logged_in' in session:
27             return test(*args, **kwargs)
28         else:
29             flash('You need to login first.')
30             return redirect(url_for('login'))
31     return wrap
32
33
34 # route handlers
35
36 @app.route('/logout/')
```



```

37 def logout():
38     session.pop('logged_in', None)
39     flash('Goodbye!')
40     return redirect(url_for('login'))
41
42
43 @app.route('/', methods=['GET', 'POST'])
44 def login():
45     if request.method == 'POST':
46         if request.form['username'] != app.config['USERNAME'] \
47             or request.form['password'] !=
48                 app.config['PASSWORD']:
49             error = 'Invalid Credentials. Please try again.'
50             return render_template('login.html', error=error)
51         else:
52             session['logged_in'] = True
53             flash('Welcome!')
54             return redirect(url_for('tasks'))
55     return render_template('login.html')

```

Save the file.

What's happening?

You've seen this all before, in the *Flask Blog App* chapter.

1. Right now, we have one view, *login.html*, which is mapped to the main URL, */*.
2. Sessions are setup, 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 setup. 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.

Please refer to the blog application from the Flask Blog App chapter for further explanation on any details of this code that you do not understand.**

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

Templates and Styles

Login template (child)

```
1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr.</h1>
5 <h3>Please login to access your task list.</h3>
6
7 <form method="post" action="/">
8   <span>Username: <input type="text" name="username"
9     value="{{request.form.username }}"></span>
10   <span>Password: <input type="password" name="password"
11     value="{{request.form.password }}"></span>
12   <input type="submit" value="Login">
13 </form>
14
15 <p><em>Use 'admin' for the username and password.</em></p>
16
17 {% endblock %}
```

Save this as *login.html* in the “templates” directory.

Base template (parent)

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Welcome to FlaskTaskr!!</title>
5     <link rel="stylesheet" href="{{ url_for('static',
6       filename='css/main.css') }}">
7   </head>
8   <body>
9     <div class="page">
10
11     {% for message in get_flashed_messages() %}
12       <div class="flash">{{ message }}</div>
13     {% endfor %}
```

```

14     {% if error %}
15         <div class="error"><strong>Error:</strong> {{ error }}</div>
16     {% endif %}
17
18     <br>
19
20     {% block content %}
21     {% endblock %}
22
23 </div>
24 </body>
25 </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 even more information.

Stylesheet

For now we’ll temporarily “borrow” the majority of the stylesheet from the Flask [tutorial](#). Please copy and paste this. Save this as `main.css` in the “css” directory within the “static” directory.

```

1 body {
2     font-family: sans-serif;
3     background: #eee;
4 }
5
6 a, h1, h2 {
7     color: #377BA8;
8 }
9
10 h1, h2 {
11     font-family: 'Georgia', serif;
12     margin: 0;
13 }
14
15 h1 {
16     border-bottom: 2px solid #eee;
17 }
18

```

```

19 h2 {
20   font-size: 1.5em;
21 }
22
23 .page {
24   margin: 2em auto;
25   width: 50em;
26   border: 5px solid #ccc;
27   padding: 0.8em;
28   background: white;
29 }
30
31 .entries {
32   list-style: none;
33   margin: 0;
34   padding: 0;
35 }
36
37 .entries li {
38   margin: 0.8em 1.2em;
39 }
40
41 .entries li h2 {
42   margin-left: -1em;
43 }
44
45 .add-task {
46   font-size: 0.9em;
47   border-bottom: 1px solid #ccc;
48 }
49 .add-task dl {
50   font-weight: bold;
51 }
52
53 .metanav {
54   text-align: right;
55   font-size: 0.8em;
56   padding: 0.3em;
57   margin-bottom: 1em;
58   background: #fafafa;

```

```

59 }
60
61 .flash {
62     background: #CEE5F5;
63     padding: 0.5em;
64 }
65
66 .error {
67     background: #F0D6D6;
68     padding: 0.5em;
69 }
70
71 .datagrid table {
72     border-collapse: collapse;
73     text-align: left;
74     width: 100%;
75 }
76
77 .datagrid {
78     background: #fff;
79     overflow: hidden;
80     border: 1px solid #000000;
81     border-radius: 3px;
82 }
83
84 .datagrid table td, .datagrid table th {
85     padding: 3px 10px;
86 }
87
88 .datagrid table thead th {
89     background-color: #000000;
90     color: #FFFFFF;
91     font-size: 15px;
92     font-weight: bold;
93 }
94 .datagrid table thead th:first-child {
95     border: none;
96 }
97
98 .datagrid table tbody td {

```

```

99     color: #000000;
100    border-left: 1px solid #E1EEF4;
101    font-size: 12px;
102    font-weight: normal;
103 }
104
105 .datagrid table tbody .alt td {
106     background: #E1EEF4;
107     color: #000000;
108 }
109
110 .datagrid table tbody td:first-child {
111     border-left: none;
112 }
113
114 .datagrid table tbody tr:last-child td {
115     border-bottom: none;
116 }
117
118 .button {
119     background-color: #000;
120     display: inline-block;
121     color: #ffffff;
122     font-size: 13px;
123     padding: 3px 12px;
124     margin: 0;
125     text-decoration: none;
126     position: relative;
127 }

```

Test

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.

```
1 # project/run.py
2
3
4 from views import app
5 app.run(debug=True)
```

Save this as *run.py* in your “project” directory. Your project structure, within the *project* directory, should now look like this:

```
1
2 _config.py
3 db_create.py
4 flasktaskr.db
5 run.py
6 static
7     css
8         main.css
9     img
10    js
11 templates
12     _base.html
13     login.html
14 views.py
```

Fire up the server:

```
1 $ 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 setup the *tasks.html* page yet. Essentially, we're ensuring that the-

- App runs,
- Templates are working correctly, and
- Basic logic we coded in *views.py* works

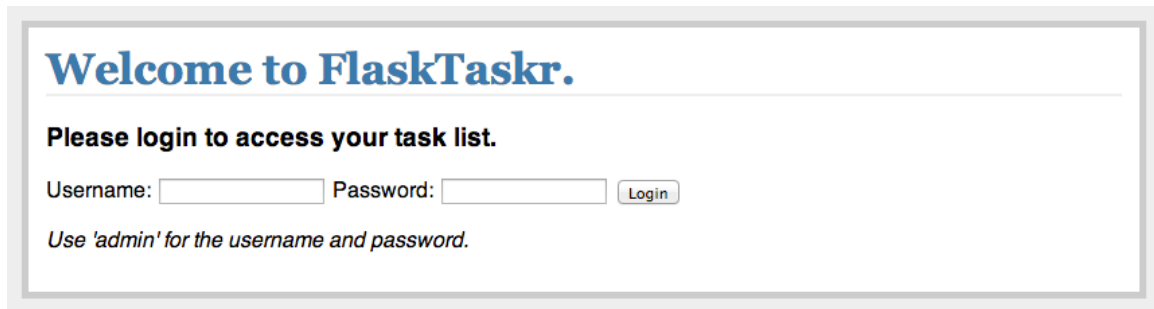


Figure 9.1: Flasktaskr login page

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 the Flask server

This should be much easier to follow vs. when all those responsibilities were crammed into one single file.

Now is a good time to commit and PUSH your code up to Github. Just make sure to add a *.gitignore* file to the "flasktaskr" directory:

```
1 env
2 venv
3 *.pyc
4 __pycache__
5 *.DS_Store
```


Tasks

The *tasks.html* page will be quite a bit different from the *main.html* page from the blog app as the user will have full CRUD access. In other words, the user will have the ability to delete tasks (delete) and mark tasks as complete (update) rather than just being able to add new tasks (create) to the database table and view (read) such tasks.

Let's start by adding the following route handler and view function to the *views.py* file:

```
1 @app.route('/tasks/')
2 @login_required
3 def tasks():
4     g.db = connect_db()
5     cur = g.db.execute(
6         'select name, due_date, priority, task_id from tasks where
7           status=1'
8     )
9     open_tasks = [
10         dict(name=row[0], due_date=row[1], priority=row[2],
11             task_id=row[3]) for row in cur.fetchall()
12     ]
13     cur = g.db.execute(
14         'select name, due_date, priority, task_id from tasks where
15           status=0'
16     )
17     closed_tasks = [
18         dict(name=row[0], due_date=row[1], priority=row[2],
19             task_id=row[3]) for row in cur.fetchall()
20     ]
21     g.db.close()
22     return render_template(
23         'tasks.html',
24         form=AddTaskForm(request.form),
25         open_tasks=open_tasks,
26         closed_tasks=closed_tasks
27     )
```

Make sure to add *g* to the imports:

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

What's happening?

We're querying the database for open and closed tasks (read), assigning the results to two variables, `open_tasks` and `closed_tasks`, and then passing 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-

```
1 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 (create), mark tasks as complete (update), and delete tasks (delete). Add each of these three functions to the *views.py* file:

```
1 # Add new tasks
2 @app.route('/add/', methods=['POST'])
3 @login_required
4 def new_task():
5     g.db = connect_db()
6     name = request.form['name']
7     date = request.form['due_date']
8     priority = request.form['priority']
9     if not name or not date or not priority:
10         flash("All fields are required. Please try again.")
11         return redirect(url_for('tasks'))
12     else:
13         g.db.execute('insert into tasks (name, due_date, priority,
14                        status) \
15                        values (?, ?, ?, 1)', [
16                         request.form['name'],
17                         request.form['due_date'],
18                         request.form['priority']
19                     ])
20         g.db.commit()
21         g.db.close()
22         flash('New entry was successfully posted. Thanks.')
23         return redirect(url_for('tasks'))
24
25
26 # Mark tasks as complete
27 @app.route('/complete/<int:task_id>/')
28 @login_required
29 def complete(task_id):
30     g.db = connect_db()
31     g.db.execute(
32         'update tasks set status = 0 where task_id='+str(task_id)
33     )
34     g.db.commit()
```

```

35     g.db.close()
36     flash('The task was marked as complete.')
37     return redirect(url_for('tasks'))
38
39
40 # Delete Tasks
41 @app.route('/delete/<int:task_id>/')
42 @login_required
43 def delete_entry(task_id):
44     g.db = connect_db()
45     g.db.execute('delete from tasks where task_id='+str(task_id))
46     g.db.commit()
47     g.db.close()
48     flash('The task was deleted.')
49     return redirect(url_for('tasks'))

```

What's happening?

1. 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 in a previous chapter. Read more about it [here](#).

Tasks Template

```
1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr</h1>
5 <a href="/logout">Logout</a>
6 <div class="add-task">
7   <h3>Add a new task:</h3>
8   <table>
9     <tr>
10       <form action="{% url_for('new_task') %}" method="post">
11         <td>
12           <label>Task Name:</label>
13           <input name="name" type="text">
14         </td>
15         <td>
16           <label>Due Date (mm/dd/yyyy):</label>
17           <input name="due_date" type="text" width="120px">
18         </td>
19         <td>
20           <label>Priority:</label>
21           <select name="priority" width="100px">
22             <option value="1">1</option>
23             <option value="2">2</option>
24             <option value="3">3</option>
25             <option value="4">4</option>
26             <option value="5">5</option>
27             <option value="6">6</option>
28             <option value="7">7</option>
29             <option value="8">8</option>
30             <option value="9">9</option>
31             <option value="10">10</option>
32           </select>
33         </td>
34         <td>
35           &nbsp;
36           &nbsp;
37           <input class="button" type="submit" value="Save">
```

```

38         </td>
39     </form>
40 </tr>
41 </table>
42 </div>
43 <div class="entries">
44     <br>
45     <br>
46     <h2>Open tasks:</h2>
47     <div class="datagrid">
48         <table>
49             <thead>
50                 <tr>
51                     <th width="300px"><strong>Task Name</strong></th>
52                     <th width="100px"><strong>Due Date</strong></th>
53                     <th width="50px"><strong>Priority</strong></th>
54                     <th><strong>Actions</strong></th>
55                 </tr>
56             </thead>
57             {% for task in open_tasks %}
58                 <tr>
59                     <td width="300px">{{ task.name }}</td>
60                     <td width="100px">{{ task.due_date }}</td>
61                     <td width="50px">{{ task.priority }}</td>
62                     <td>
63                         <a href="{% url_for('delete_entry', task_id =
64                             task.task_id) %}">Delete</a> -
65                         <a href="{% url_for('complete', task_id = task.task_id)
66                             %}">Mark as Complete</a>
67                     </td>
68                 </tr>
69             {% endfor %}
70         </table>
71     </div>
72     <br>
73     <br>
74     </div>
75     <div class="entries">
76         <h2>Closed tasks:</h2>
77         <div class="datagrid">

```

```

76 <table>
77   <thead>
78     <tr>
79       <th width="300px"><strong>Task Name</strong></th>
80       <th width="100px"><strong>Due Date</strong></th>
81       <th width="50px"><strong>Priority</strong></th>
82       <th><strong>Actions</strong></th>
83     </tr>
84   </thead>
85   {% for task in closed_tasks %}
86     <tr>
87       <td width="300px">{{ task.name }}</td>
88       <td width="100px">{{ task.due_date }}</td>
89       <td width="50px">{{ task.priority }}</td>
90       <td>
91         <a href="{% url_for('delete_entry', task_id =
          task.task_id) %}">Delete</a>
92       </td>
93     </tr>
94   {% endfor %}
95 </table>
96 </div>
97 </div>
98
99 {% endblock %}

```

Save this as *tasks.html* in the “templates” directory.

What’s happening?

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

```

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

```

Essentially, we pull the `task_id` from the database dynamically from each row in the database table as the for loop progresses (or iterates). We then assign it to a variable, also named `task_id`, which is then passed back to either the `delete_entry()` func-

tion - `@app.route('/delete/<int:task_id>')` - or the `complete()` function - `@app.route('/complete/<int:task_id>')`.

Make sure to walk through this app line by line. You should understand what each line is doing. Add comments to help you remember what's happening.

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:

```
1 $ pip install Flask-WTF==0.11
2 $ pip freeze > requirements.txt
```

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

```
1 # project/forms.py
2
3
4 from flask_wtf import Form
5 from wtforms import StringField, DateField, IntegerField, \
6     SelectField
7 from wtforms.validators import DataRequired
8
9
10 class AddTaskForm(Form):
11     task_id = IntegerField()
12     name = StringField('Task Name', validators=[DataRequired()])
13     due_date = DateField(
14         'Date Due (mm/dd/yyyy)',
15         validators=[DataRequired()], format='%m/%d/%Y'
16     )
17     priority = SelectField(
18         'Priority',
19         validators=[DataRequired()],
20         choices=[
21             ('1', '1'), ('2', '2'), ('3', '3'), ('4', '4'), ('5',
22                 '5'),
23             ('6', '6'), ('7', '7'), ('8', '8'), ('9', '9'), ('10',
24                 '10')
25         ]
26     )
27     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. For example, `Required` 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 are setup 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`:

```
1 from forms import AddTaskForm
```

Test

Finally, test out the functionality of the app.

Fire up your server again. You should be able to login 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.

Welcome to FlaskTaskR
[Logout](#)

Add a new task:

Task Name: Due Date (mm/dd/yyyy): Priority:

Open tasks:

Task Name	Due Date	Priority	Actions
Finish this tutorial	02/03/2013	10	Delete - Mark as Complete
Finish my book	02/03/2013	10	Delete - Mark as Complete

Closed tasks:

Task Name	Due Date	Priority	Actions
Take a shower	3/26/2012	5	Delete

Figure 9.2: Flasktaskr tasks page

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

Your project structure within "project" should look like this:

```
1
2 _config.py
3 db_create.py
4 flasktaskr.db
5 forms.py
6 run.py
7 static
8     css
9         main.css
10    img
```

```
11     js
12 templates
13     _base.html
14     login.html
15     tasks.html
16 views.py
```

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

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

Chapter 10

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
Unit 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

Task	Complete
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 login 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:

```
1 $ pip install Flask-SQLAlchemy==2.0
2 $ 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.

```
1 # project/models.py
2
3
4 from views import db
```

```

5
6
7 class Task(db.Model):
8
9     __tablename__ = "tasks"
10
11     task_id = db.Column(db.Integer, primary_key=True)
12     name = db.Column(db.String, nullable=False)
13     due_date = db.Column(db.Date, nullable=False)
14     priority = db.Column(db.Integer, nullable=False)
15     status = db.Column(db.Integer)
16
17     def __init__(self, name, due_date, priority, status):
18         self.name = name
19         self.due_date = due_date
20         self.priority = priority
21         self.status = status
22
23     def __repr__(self):
24         return '<name {0}>'.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*:

```

1 from forms import AddTaskForm
2
3 from functools import wraps
4 from flask import Flask, flash, redirect, render_template, \
5     request, session, url_for
6 from flask.ext.sqlalchemy import SQLAlchemy
7
8 # config
9
10 app = Flask(__name__)
11 app.config.from_object('_config')
12 db = SQLAlchemy(app)
13
14 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:

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

Update the route handlers in *views.py*:

```
1 @app.route('/logout/')
2 def logout():
3     session.pop('logged_in', None)
4     flash('Goodbye!')
5     return redirect(url_for('login'))
6
7
8 @app.route('/', methods=['GET', 'POST'])
9 def login():
10     error = None
11     if request.method == 'POST':
12         if request.form['username'] != app.config['USERNAME'] or \
13             request.form['password'] != app.config['PASSWORD']:
14             error = 'Invalid Credentials. Please try again.'
15             return render_template('login.html', error=error)
16         else:
17             session['logged_in'] = True
18             flash('Welcome!')
19             return redirect(url_for('tasks'))
20     return render_template('login.html')
21
22
23 @app.route('/tasks/')
24 @login_required
25 def tasks():
26     open_tasks = db.session.query(Task) \
27         .filter_by(status='1').order_by(Task.due_date.asc())
28     closed_tasks = db.session.query(Task) \
29         .filter_by(status='0').order_by(Task.due_date.asc())
30     return render_template(
31         'tasks.html',
32         form=AddTaskForm(request.form),
```

```

33         open_tasks=open_tasks,
34         closed_tasks=closed_tasks
35     )
36
37
38 @app.route('/add/', methods=['GET', 'POST'])
39 @login_required
40 def new_task():
41     form = AddTaskForm(request.form)
42     if request.method == 'POST':
43         if form.validate_on_submit():
44             new_task = Task(
45                 form.name.data,
46                 form.due_date.data,
47                 form.priority.data,
48                 '1'
49             )
50             db.session.add(new_task)
51             db.session.commit()
52             flash('New entry was successfully posted. Thanks.')
53     return redirect(url_for('tasks'))
54
55
56 @app.route('/complete/<int:task_id>/')
57 @login_required
58 def complete(task_id):
59     new_id = task_id
60     db.session.query(Task).filter_by(task_id=new_id).update({"status":
61         "0"})
62     db.session.commit()
63     flash('The task is complete. Nice.')
64     return redirect(url_for('tasks'))
65
66 @app.route('/delete/<int:task_id>/')
67 @login_required
68 def delete_entry(task_id):
69     new_id = task_id
70     db.session.query(Task).filter_by(task_id=new_id).delete()
71     db.session.commit()

```

```

72     flash('The task was deleted. Why not add a new one?')
73     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:

```

1  # project/_config.py
2
3
4  import os
5
6
7  # grab the folder where this script lives
8  basedir = os.path.abspath(os.path.dirname(__file__))
9
10 DATABASE = 'flasktaskr.db'
11 USERNAME = 'admin'
12 PASSWORD = 'admin'
13 CSRF_ENABLED = True
14 SECRET_KEY = 'my_precious'
15
16 # define the full path for the database
17 DATABASE_PATH = os.path.join(basedir, DATABASE)
18
19 # the database uri
20 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`.

```

1  # project/db_create.py
2
3
4  from views import db
5  from models import Task
6  from datetime import date
7
8

```

```

9 # create the database and the db table
10 db.create_all()
11
12 # insert data
13 db.session.add(Task("Finish this tutorial", date(2015, 3, 13), 10,
14                      1))
15 db.session.add(Task("Finish Real Python", date(2015, 3, 13), 10, 1))
16
17 # commit the changes
18 db.session.commit()

```

What's happening?

1. We initialize the database schema 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.

Create the database

Save all the files, and run the script:

```

1 $ 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:

```

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

```

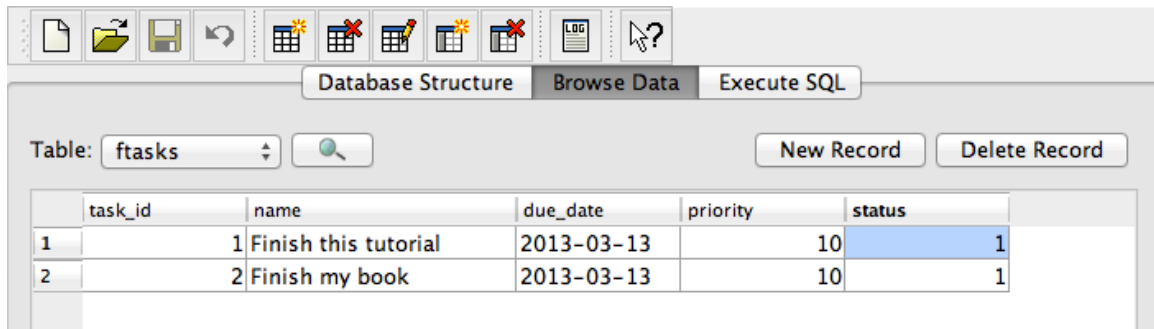


Figure 10.1: Verify data within SQLite Browser

The form should now look like this:

```

1 <form action="{ url_for('new_task') }}" method="post">
2   {{ form.csrf_token }}
3   <td>
4     <label>Task Name:</label>
5     <input name="name" type="text">
6   </td>
7   <td>
8     <label>Due Date (mm/dd/yyyy):</label>
9     <input name="due_date" type="text" width="120px">
10  </td>
11  <td>
12    <label>Priority:</label>
13    <select name="priority" width="100px">
14      <option value="1">1</option>
15      <option value="2">2</option>
16      <option value="3">3</option>
17      <option value="4">4</option>
18      <option value="5">5</option>
19      <option value="6">6</option>
20      <option value="7">7</option>
21      <option value="8">8</option>
22      <option value="9">9</option>
23      <option value="10">10</option>
24    </select>
25  </td>
26  <td>
27    &nbsp;

```

[illegible]

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*:

```
1 class User(db.Model):
2
3     __tablename__ = 'users'
4
5     id = db.Column(db.Integer, primary_key=True)
6     name = db.Column(db.String, unique=True, nullable=False)
7     email = db.Column(db.String, unique=True, nullable=False)
8     password = db.Column(db.String, nullable=False)
9
10    def __init__(self, name=None, email=None, password=None):
11        self.name = name
12        self.email = email
13        self.password = password
14
15    def __repr__(self):
16        return '<User {0}>'.format(self.name)
```

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

```
1 db.session.add(Task("Finish this tutorial", date(2015, 3, 13), 10,
2     1))
3 db.session.add(Task("Finish Real Python", date(2015, 3, 13), 10, 1))
```

If you do not do this, the script will try to 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*:

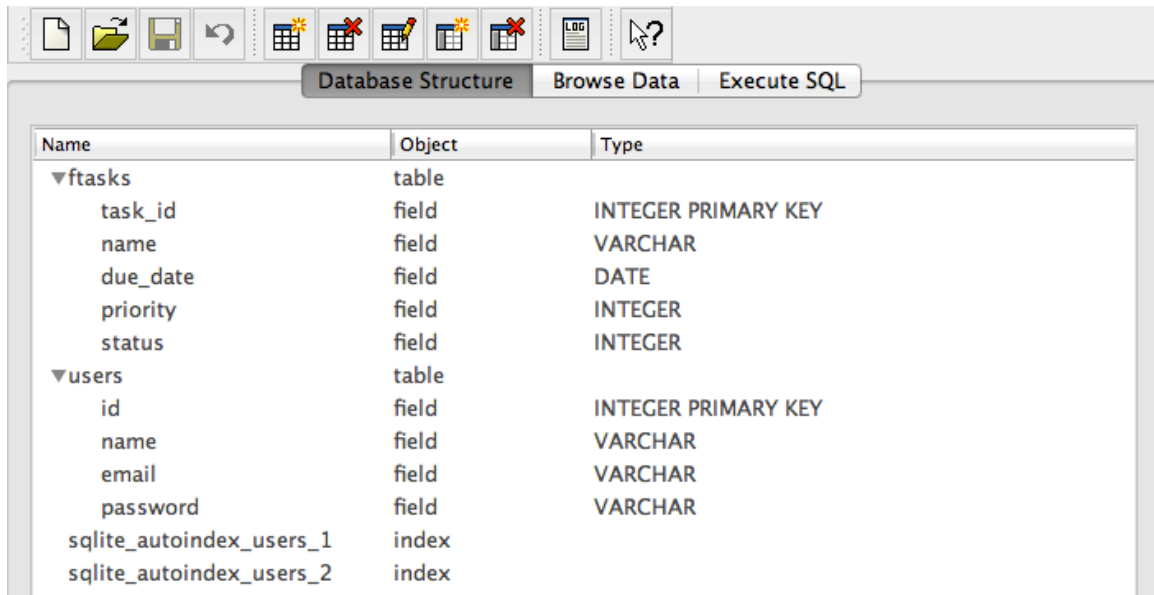


Figure 10.2: Verify data within SQLite Browser redux

```

1 USERNAME = 'admin'
2 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:

```

1 class RegisterForm(Form):
2     name = StringField(
3         'Username',
4         validators=[DataRequired(), Length(min=6, max=25)]
5     )
6     email = StringField(
7         'Email',
8         validators=[DataRequired(), Length(min=6, max=40)]
9     )
10    password = PasswordField(
11        'Password',
12        validators=[DataRequired(), Length(min=6, max=40)])
13    confirm = PasswordField(

```



```

14         'Repeat Password',
15         validators=[DataRequired(), EqualTo('password',
16             message='Passwords must match')]
17     )
18
19 class LoginForm(Form):
20     name = StringField(
21         'Username',
22         validators=[DataRequired()]
23     )
24     password = PasswordField(
25         'Password',
26         validators=[DataRequired()]
27     )

```

Then update the imports:

```

1 from flask_wtf import Form
2 from wtforms import StringField, DateField, IntegerField, \
3     SelectField, PasswordField
4 from wtforms.validators import DataRequired, Length, EqualTo

```

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

```

1 #####
2 ##### imports #####
3 #####
4
5 from forms import AddTaskForm, RegisterForm, LoginForm
6
7 from functools import wraps
8 from flask import Flask, flash, redirect, render_template, \
9     request, session, url_for
10 from flask.ext.sqlalchemy import SQLAlchemy
11
12
13 #####
14 ##### config #####
15 #####
16

```

```

17 app = Flask(__name__)
18 app.config.from_object('_config')
19 db = SQLAlchemy(app)
20
21 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():

```

1 @app.route('/register/', methods=['GET', 'POST'])
2 def register():
3     error = None
4     form = RegisterForm(request.form)
5     if request.method == 'POST':
6         if form.validate_on_submit():
7             new_user = User(
8                 form.name.data,
9                 form.email.data,
10                form.password.data,
11            )
12            db.session.add(new_user)
13            db.session.commit()
14            flash('Thanks for registering. Please login.')
15            return redirect(url_for('login'))
16    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

Registration:

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr.</h1>

```

```

5 <h3>Please register to access the task list.</h3>
6 <form method="POST" action="">
7   {{ form.csrf_token }}
8   <p>
9     {{ form.name.label }}: {{ form.name }}
10    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
11    {{ form.email.label }}: {{ form.email }}
12  </p>
13  <p>
14    {{ form.password.label }}: {{ form.password }}
15    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
16    {{ form.confirm.label }}: {{ form.confirm }}
17  </p>
18  <p><input type="submit" value="Register"></p>
19 </form>
20
21 <br>
22
23 <p><em>Already registered?</em> Click <a href="/">here</a> to
    login.</p>
24
25 {% endblock %}

```

Save this as *register.html* within your “templates” directory.

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

```
1 <br>
2
3 <p><em>Need an account? </em><a href="/register">Signup!!</a></p>
```

Be sure to remove the following code as well:

```
1 <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:

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

```
2 request.form['password'] != app.config['PASSWORD']:
```

User Login/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:

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     form = LoginForm(request.form)
5     if request.method == 'POST':
6         if form.validate_on_submit():
7             user =
8                 User.query.filter_by(name=request.form['name']).first()
9             if user is not None and user.password ==
10                 request.form['password']:
11                 session['logged_in'] = True
12                 flash('Welcome!')
13                 return redirect(url_for('tasks'))
14             else:
15                 error = 'Invalid username or password.'
16         else:
17             error = 'Both fields are required.'
18     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 following code:

```
1 {% extends "_base.html" %}
2 {% block content %}
3
```

```

4 <h1>Welcome to FlaskTaskr.</h1>
5 <h3>Please login to access your task list.</h3>
6
7 <form method="post" action="/">
8     {{ form.csrf_token }}
9     <p>
10         {{ form.name.label }}: {{ form.name }}
11         &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~\n        {{ form.password.label }}: {{ form.password }}
12         ~\n        &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~\n        <input type="submit" value="Submit">
13     </p>
14 </form>
15
16 <br>
17
18 <p><em>Need an account? </em><a href="/register">Signup!!</a></p>
19
20 {% endblock %}

```

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:

```
1 127.0.0.1 - - [26/Mar/2015 11:22:30] "POST / HTTP/1.1" 302 -
2 127.0.0.1 - - [26/Mar/2015 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: `posted_date` and `user_id` to the `tasks` table. 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:

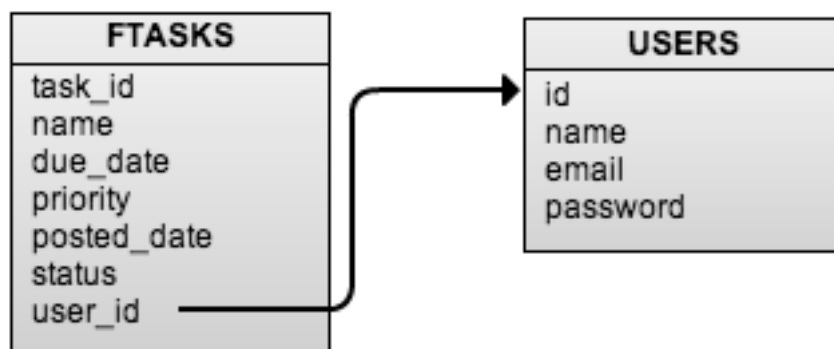


Figure 10.3: Database Relationships ERD

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

Add the following field to the “tasks” table-

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

-and this field to the “users” table:

```
1 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:

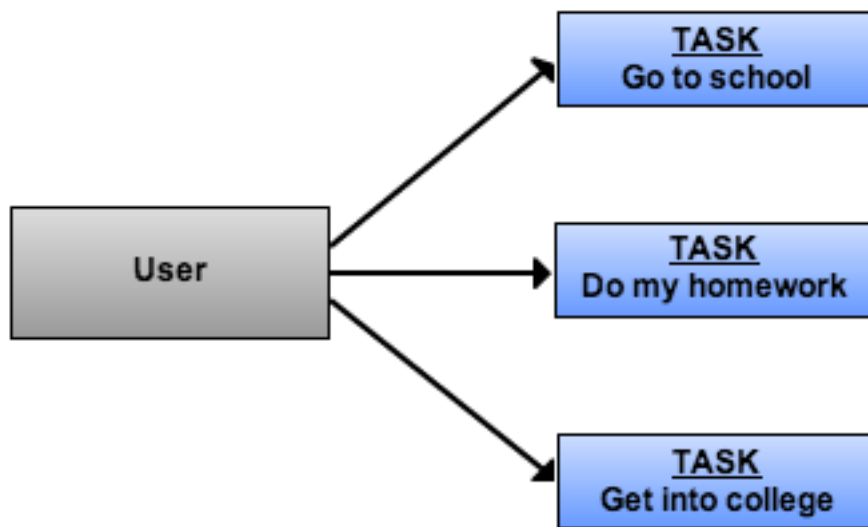


Figure 10.4: Database Relationships - one to many

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:


```

1 # project/models.py
2
3
4 from views import db
5
6 import datetime
7
8
9 class Task(db.Model):
10
11     __tablename__ = "tasks"
12
13     task_id = db.Column(db.Integer, primary_key=True)
14     name = db.Column(db.String, nullable=False)
15     due_date = db.Column(db.Date, nullable=False)
16     priority = db.Column(db.Integer, nullable=False)
17     posted_date = db.Column(db.Date,
18                             default=datetime.datetime.utcnow())
19     status = db.Column(db.Integer)
20     user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
21
22     def __init__(self, name, due_date, priority, posted_date,
23                 status, user_id):
24         self.name = name
25         self.due_date = due_date
26         self.priority = priority
27         self.posted_date = posted_date
28         self.status = status
29         self.user_id = user_id
30
31     def __repr__(self):
32         return '<name {0}>'.format(self.name)
33
34 class User(db.Model):
35
36     __tablename__ = 'users'
37
38     id = db.Column(db.Integer, primary_key=True)
39     name = db.Column(db.String, unique=True, nullable=False)

```

```

39     email = db.Column(db.String, unique=True, nullable=False)
40     password = db.Column(db.String, nullable=False)
41     tasks = db.relationship('Task', backref='poster')
42
43     def __init__(self, name=None, email=None, password=None):
44         self.name = name
45         self.email = email
46         self.password = password
47
48     def __repr__(self):
49         return '<User {0}>'.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:

```

1  # project/db_migrate.py
2
3
4  from views import db
5  from _config import DATABASE_PATH
6
7  import sqlite3
8  from datetime import datetime
9
10 with sqlite3.connect(DATABASE_PATH) as connection:
11
12     # get a cursor object used to execute SQL commands
13     c = connection.cursor()
14
15     # temporarily change the name of tasks table
16     c.execute("""ALTER TABLE tasks RENAME TO old_tasks""")
17
18     # recreate a new tasks table with updated schema
19     db.create_all()
20
21     # retrieve data from old_tasks table
22     c.execute("""SELECT name, due_date, priority,
23                 status FROM old_tasks ORDER BY task_id ASC""")
24

```

```

25     # save all rows as a list of tuples; set posted_date to now and
        user_id to 1
26     data = [(row[0], row[1], row[2], row[3],
27              datetime.now(), 1) for row in c.fetchall()]
28
29     # insert data to tasks table
30     c.executemany("""INSERT INTO tasks (name, due_date, priority,
        status,
31                  posted_date, user_id) VALUES (?, ?, ?, ?, ?,
        ?)""", data)
32
33     # delete old_tasks table
34     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.

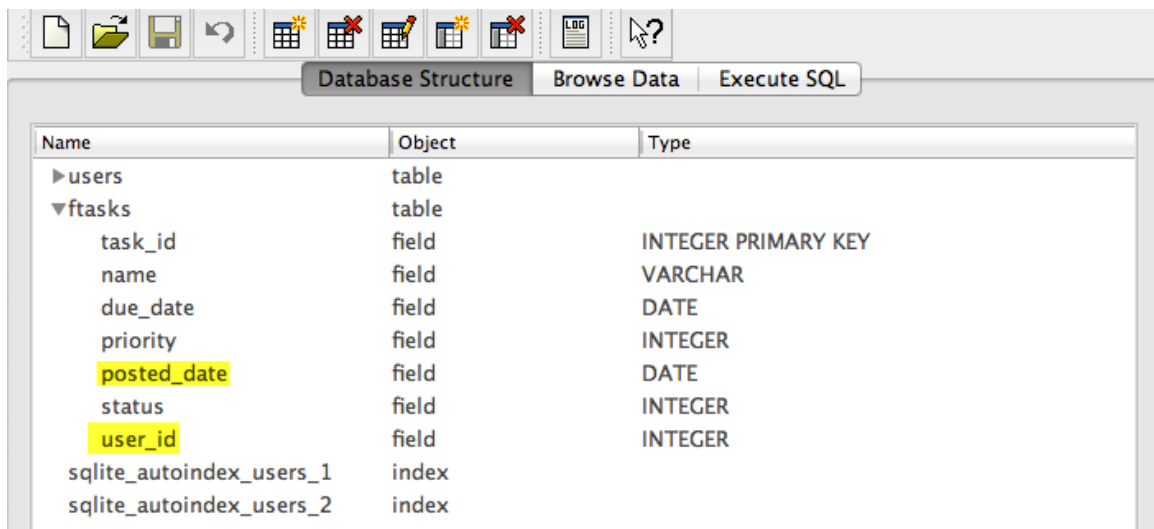


Figure 10.5: Verify updated schema within SQLite Browser

Controller

We also need to update the adding of tasks within *views.py*. Within the *new_task* function, change the following:

```

1 new_task = Task(
2     form.name.data,
3     form.due_date.data,
4     form.priority.data,
5     '1'
6 )

```

to:

```

1 new_task = Task(
2     form.name.data,
3     form.due_date.data,
4     form.priority.data,
5     datetime.datetime.utcnow(),
6     '1',
7     '1'
8 )

```

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:

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr</h1>
5 <br>
6 <a href="/logout">Logout</a>
7 <div class="add-task">
8     <h3>Add a new task:</h3>
9     <form action="{{ url_for('new_task') }}" method="post">
10         {{ form.csrf_token }}
11     <p>

```

```

12         {{ form.name.label }}: {{ form.name }}<br>
13         {{ form.due_date.label }}: {{ form.due_date }}<br>
14         {{ form.priority.label }}: {{ form.priority }}
15     </p>
16     <p><input type="submit" value="Submit"></p>
17 </form>
18 </div>
19 <div class="entries">
20     <br>
21     <br>
22     <h2>Open tasks:</h2>
23     <div class="datagrid">
24         <table>
25             <thead>
26                 <tr>
27                     <th width="200px"><strong>Task Name</strong></th>
28                     <th width="75px"><strong>Due Date</strong></th>
29                     <th width="100px"><strong>Posted Date</strong></th>
30                     <th width="50px"><strong>Priority</strong></th>
31                     <th width="90px"><strong>Posted By</strong></th>
32                     <th><strong>Actions</strong></th>
33                 </tr>
34             </thead>
35             {% for task in open_tasks %}
36                 <tr>
37                     <td width="200px">{{ task.name }}</td>
38                     <td width="75px">{{ task.due_date }}</td>
39                     <td width="100px">{{ task.posted_date }}</td>
40                     <td width="50px">{{ task.priority }}</td>
41                     <td width="90px">{{ task.poster.name }}</td>
42                     <td>
43                         <a href="{{ url_for('delete_entry', task_id =
44                             task.task_id) }}">Delete</a> -
45                         <a href="{{ url_for('complete', task_id = task.task_id)
46                             }}">Mark as Complete</a>
47                     </td>
48                 </tr>
49             {% endfor %}
50         </table>
51     </div>

```

```

50 </div>
51 <br>
52 <br>
53 <div class="entries">
54   <h2>Closed tasks:</h2>
55   <div class="datagrid">
56     <table>
57       <thead>
58         <tr>
59           <th width="200px"><strong>Task Name</strong></th>
60           <th width="75px"><strong>Due Date</strong></th>
61           <th width="100px"><strong>Posted Date</strong></th>
62           <th width="50px"><strong>Priority</strong></th>
63           <th width="90px"><strong>Posted By</strong></th>
64           <th><strong>Actions</strong></th>
65         </tr>
66       </thead>
67       {% for task in closed_tasks %}
68         <tr>
69           <td width="200px">{{ task.name }}</td>
70           <td width="75px">{{ task.due_date }}</td>
71           <td width="100px">{{ task.posted_date }}</td>
72           <td width="50px">{{ task.priority }}</td>
73           <td width="90px">{{ task.poster.name }}</td>
74           <td>
75             <a href="{% url_for('delete_entry', task_id =
              task.task_id) %}">Delete</a>
76           </td>
77         </tr>
78       {% endfor %}
79     </table>
80   </div>
81 <div>
82
83 {% endblock %}

```

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
Can I still add?	2013-02-04	2013-03-26	1	michael	Delete - Mark as Complete
different user	2013-02-04	2013-02-02	1	michael	Delete - Mark as Complete
Finish this tutorial	2013-03-13	2013-03-26	10	michael	Delete - Mark as Complete
Finish my book	2013-03-13	2013-03-26	10	michael	Delete - Mark as Complete
Finish this tutorial	2013-03-13	2013-03-26	10	michael	Delete - Mark as Complete
Finish my book	2013-03-13	2013-03-26	10	michael	Delete - Mark as Complete

Figure 10.6: Flasktaskr tasks page showing just one user

Let's correct that.

Managing Sessions

Do you remember the relationship we established between the two tables in the last lesson?

```
1 user_id = Column(Integer, ForeignKey('users.id'))
2 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`:

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     form = LoginForm(request.form)
5     if request.method == 'POST':
6         if form.validate_on_submit():
7             user =
8                 User.query.filter_by(name=request.form['name']).first()
9             if user is not None and user.password ==
10                 request.form['password']:
11                 session['logged_in'] = True
12                 session['user_id'] = user.id
13                 flash('Welcome!')
14                 return redirect(url_for('tasks'))
15             else:
16                 error = 'Invalid username or password.'
17         else:
18             error = 'Both fields are required.'
19     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:

```
1 @app.route('/add/', methods=['GET', 'POST'])
2 @login_required
3 def new_task():
4     form = AddTaskForm(request.form)
5     if request.method == 'POST':
6         if form.validate_on_submit():
7             new_task = Task(
8                 form.name.data,
```



```

9         form.due_date.data,
10        form.priority.data,
11        datetime.datetime.utcnow(),
12        '1',
13        session['user_id']
14    )
15    db.session.add(new_task)
16    db.session.commit()
17    flash('New entry was successfully posted. Thanks.')
18    return redirect(url_for('tasks'))
19 else:
20     flash('All fields are required.')
21     return redirect(url_for('tasks'))
22 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:

```

1 @app.route('/logout/')
2 def logout():
3     session.pop('logged_in', None)
4     session.pop('user_id', None)
5     flash('Goodbye!')
6     return redirect(url_for('login'))

```

Now open up `tasks.html`. In each of the two for loops, note these statements:

```

1 <td width="90px">{{ task.poster.name }}</td>

```

and

```

1 <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:

```
1
2  _config.py
3  db_create.py
4  db_migrate.py
5  flasktaskr.db
6  forms.py
7  models.py
8  run.py
9  static
10     css
11         main.css
12     img
13     js
14 templates
15     _base.html
16     login.html
17     register.html
18     tasks.html
19 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-o2* folder in the course [repository](#).

Chapter 11

Flask: FlaskTaskr, Part 3 - Error Handling and Unit Testing

Welcome back!

Let's see where we're at:

Task	Complete
Database Management	Yes
User Registration	Yes
User Login/Authentication	Yes
Database Relationships	Yes
Managing Sessions	Yes
Error Handling	No
Unit 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

Task	Complete
Boilerplate and Workflow	No
Continuous Integration and Delivery	No

Let's get to it. First, make sure your app is working properly. Register a new user, login, 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 unit testing. You will also be introduced to a concept/development paradigm known as Test Driven Development.

Error Handling

Error handling is a means of dealing with errors should they occur at runtime - e.g., when the application is executing a task.

Let's look at an error.

Form Validation Errors

Try to register a new user without entering any information. Nothing should happen - and nothing does. Literally. This is confusing for the end user. Thus, we need to add in an error message in order to provide good feedback so that the user knows how to proceed. Fortunately WTForms provides error messages for any form that has a validator attached to it.

Open *forms.py*. There are already some validators in place; it's pretty straightforward. For example, in the `RegisterForm()` class, the *name* field should be between 6 and 25 characters:

```
1 class RegisterForm(Form):
2     name = StringField(
3         'Username',
4         validators=[DataRequired(), Length(min=6, max=25)]
5     )
6     email = StringField(
7         'Email',
8         validators=[DataRequired(), Length(min=6, max=40)]
9     )
10    password = PasswordField(
11        'Password',
12        validators=[DataRequired(), Length(min=6, max=40)])
13    confirm = PasswordField(
14        'Repeat Password',
15        validators=[DataRequired(), EqualTo('password',
16            message='Passwords must match')]
```

Let's add a few more:

```
1 class RegisterForm(Form):
2     name = StringField(
3         'Username',
4         validators=[DataRequired(), Length(min=6, max=25)]
```

```

5     )
6     email = StringField(
7         'Email',
8         validators=[DataRequired(), Email(), Length(min=6, max=40)]
9     )
10    password = PasswordField(
11        'Password',
12        validators=[DataRequired(), Length(min=6, max=40)])
13    confirm = PasswordField(
14        'Repeat Password',
15        validators=[DataRequired(), EqualTo('password')]
16    )

```

Be sure to update the imports as well:

```

1 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:

```

1 def flash_errors(form):
2     for field, errors in form.errors.items():
3         for error in errors:
4             flash(u"Error in the %s field - %s" % (
5                 getattr(form, field).label.text, error), 'error')

```

Then update the templates:

register.html

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr.</h1>
5 <h3>Please register to access the task list.</h3>
6 <form method="POST" action="/register/">
7     {{ form.csrf_token }}
8     <p>
9         {{ form.name(placeholder="name") }}
10        {% if form.name.errors %}
11            <span class="error">

```

```

12         {% for error in form.name.errors %}
13             {{ error }}
14         {% endfor %}
15     </span>
16 {% endif %}
17 </p>
18 <p>
19     {{ form.email(placeholder="email address") }}
20     {% if form.email.errors %}
21         <span class="error">
22             {% for error in form.email.errors %}
23                 {{ error }}
24             {% endfor %}
25         </span>
26     {% endif %}
27 </p>
28 <p>
29     {{ form.password(placeholder="password") }}
30     {% if form.password.errors %}
31         <span class="error">
32             {% for error in form.password.errors %}
33                 {{ error }}
34             {% endfor %}
35         </span>
36     {% endif %}
37 </p>
38 <p>
39     {{ form.confirm(placeholder="confirm password") }}
40     {% if form.confirm.errors %}
41         <span class="error">
42             {% for error in form.confirm.errors %}
43                 {{ error }}
44             {% endfor %}
45         </span>
46     {% endif %}
47 </p>
48 <button class="btn btn-sm btn-success"
49     type="submit">Register</button>
50 <br>
51 <p><em>Already registered?</em> Click <a href="/">here</a> to

```

```

        login.</p>
51 </form>
52
53 {% endblock %}

```

Then update the CSS styles associated with the error class:

```

1 .error {
2     color: red;
3     font-size: .8em;
4     background-color: #FFFFFF;
5 }

```

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.

Update the remaining two templates...

login.html

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr.</h1>
5 <div class="lead">Please sign in to access your task list</div>
6 <form class="form-signin" role="form" method="post" action="/">
7     {{ form.csrf_token }}
8     <p>
9         {{ form.name(placeholder="name") }}
10        <span class="error">
11            {% if form.name.errors %}
12                {% for error in form.name.errors %}
13                    {{ error }}
14                {% endfor %}
15            {% endif %}
16        </span>
17    </p>
18    <p>
19        {{ form.password(placeholder="password") }}
20        <span class="error">

```



```

21     {% if form.password.errors %}
22         {% for error in form.password.errors %}
23             {{ error }}
24         {% endfor %}
25     {% endif %}
26 </span>
27 </p>
28 <button class="btn btn-sm btn-success" type="submit">Sign
    in</button>
29 <br>
30 <br>
31 <p><em>Need an account? </em><a href="/register">Signup!!</a></p>
32 </form>
33
34 {% endblock %}

```

tasks.html

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr</h1>
5 <br>
6 <a href="/logout">Logout</a>
7 <div class="add-task">
8     <h3>Add a new task:</h3>
9     <form action="{% url_for('new_task') %}" method="post">
10         {{ form.csrf_token }}
11         <p>
12             {{ form.name(placeholder="name") }}
13             <span class="error">
14                 {% if form.name.errors %}
15                     {% for error in form.name.errors %}
16                         {{ error }}
17                     {% endfor %}
18                 {% endif %}
19             </span>
20         </p>
21         <p>
22             {{ form.due_date(placeholder="due date") }}

```

```

23     <span class="error">
24         {% if form.due_date.errors %}
25             {% for error in form.due_date.errors %}
26                 {{ error }}
27             {% endfor %}
28         {% endif %}
29     </span>
30 </p>
31 <p>
32     {{ form.priority.label }}
33     {{ form.priority }}
34     <span class="error">
35         {% if form.priority.errors %}
36             {% for error in form.priority.errors %}
37                 {{ error }}
38             {% endfor %}
39         {% endif %}
40     </span>
41 </p>
42 <p><input type="submit" value="Submit"></p>
43 </form>
44 </div>
45 <div class="entries">
46     <br>
47     <br>
48     <h2>Open tasks:</h2>
49     <div class="datagrid">
50         <table>
51             <thead>
52                 <tr>
53                     <th width="200px"><strong>Task Name</strong></th>
54                     <th width="75px"><strong>Due Date</strong></th>
55                     <th width="100px"><strong>Posted Date</strong></th>
56                     <th width="50px"><strong>Priority</strong></th>
57                     <th width="90px"><strong>Posted By</strong></th>
58                     <th><strong>Actions</strong></th>
59                 </tr>
60             </thead>
61             {% for task in open_tasks %}
62                 <tr>

```

```

63         <td width="200px">{{ task.name }}</td>
64         <td width="75px">{{ task.due_date }}</td>
65         <td width="100px">{{ task.posted_date }}</td>
66         <td width="50px">{{ task.priority }}</td>
67         <td width="90px">{{ task.poster.name }}</td>
68         <td>
69             <a href="{{ url_for('delete_entry', task_id =
70                 task.task_id) }}">Delete</a> -
71             <a href="{{ url_for('complete', task_id = task.task_id)
72                 }}">Mark as Complete</a>
73         </td>
74     </tr>
75     {% endfor %}
76 </table>
77 </div>
78 </div>
79 <br>
80 <br>
81 <div class="entries">
82     <h2>Closed tasks:</h2>
83     <div class="datagrid">
84         <table>
85             <thead>
86                 <tr>
87                     <th width="200px"><strong>Task Name</strong></th>
88                     <th width="75px"><strong>Due Date</strong></th>
89                     <th width="100px"><strong>Posted Date</strong></th>
90                     <th width="50px"><strong>Priority</strong></th>
91                     <th width="90px"><strong>Posted By</strong></th>
92                     <th><strong>Actions</strong></th>
93                 </tr>
94             </thead>
95             {% for task in closed_tasks %}
96                 <tr>
97                     <td width="200px">{{ task.name }}</td>
98                     <td width="75px">{{ task.due_date }}</td>
99                     <td width="100px">{{ task.posted_date }}</td>
100                    <td width="50px">{{ task.priority }}</td>

```

```

101         <a href="{ url_for('delete_entry', task_id =
102             task.task_id) }" >Delete</a>
103     </td>
104 </tr>
105 {% endfor %}
106 </table>
107 </div>
108 <div>
109 {% endblock %}

```

Update the view as well:

```

1 @app.route('/add/', methods=['GET', 'POST'])
2 @login_required
3 def new_task():
4     error = None
5     form = AddTaskForm(request.form)
6     if request.method == 'POST':
7         if form.validate_on_submit():
8             new_task = Task(
9                 form.name.data,
10                form.due_date.data,
11                form.priority.data,
12                datetime.datetime.utcnow(),
13                '1',
14                session['user_id']
15            )
16            db.session.add(new_task)
17            db.session.commit()
18            flash('New entry was successfully posted. Thanks.')
19            return redirect(url_for('tasks'))
20         else:
21             return render_template('tasks.html', form=form,
22                                   error=error)
23     return render_template('tasks.html', form=form, error=error)

```

Test this out.

Did you notice that the errors display, but that the open and closed tasks don't show up? This is because we are not passing them in.

To update that, let's create helper functions out of the opening and closing of tasks.

```
1 def open_tasks():
2     return db.session.query(Task).filter_by(
3         status='1').order_by(Task.due_date.asc())
4
5
6 def closed_tasks():
7     return db.session.query(Task).filter_by(
8         status='0').order_by(Task.due_date.asc())
```

Now update tasks() and new_task():

```
1 @app.route('/tasks/')
2 @login_required
3 def tasks():
4     return render_template(
5         'tasks.html',
6         form=AddTaskForm(request.form),
7         open_tasks=open_tasks(),
8         closed_tasks=closed_tasks()
9     )
10
11 @app.route('/add/', methods=['GET', 'POST'])
12 @login_required
13 def new_task():
14     error = None
15     form = AddTaskForm(request.form)
16     if request.method == 'POST':
17         if form.validate_on_submit():
18             new_task = Task(
19                 form.name.data,
20                 form.due_date.data,
21                 form.priority.data,
22                 datetime.datetime.utcnow(),
23                 '1',
24                 session['user_id']
25             )
26             db.session.add(new_task)
27             db.session.commit()
28             flash('New entry was successfully posted. Thanks.')
```

```

29         return redirect(url_for('tasks'))
30     return render_template(
31         'tasks.html',
32         form=form,
33         error=error,
34         open_tasks=open_tasks(),
35         closed_tasks=closed_tasks()
36     )

```

Test it again.

Database Related Errors

Test to see what happens when you try to register someone with the same username and/or password. You should see a big, ugly IntegrityError. We need to use the try/except pair to handle the error.

First add another import to the Controller, *views.py*:

```

1 from sqlalchemy.exc import IntegrityError

```

Then update the register() function:

```

1 @app.route('/register/', methods=['GET', 'POST'])
2 def register():
3     error = None
4     form = RegisterForm(request.form)
5     if request.method == 'POST':
6         if form.validate_on_submit():
7             new_user = User(
8                 form.name.data,
9                 form.email.data,
10                form.password.data,
11            )
12            try:
13                db.session.add(new_user)
14                db.session.commit()
15                flash('Thanks for registering. Please login.')
16                return redirect(url_for('login'))
17            except IntegrityError:
18                error = 'That username and/or email already exist.'
19                return render_template('register.html', form=form,
30                                     error=error)

```

```
return render_template('register.html', form=form, error=error)
```

Essentially, the code within the `try` block attempts to execute. If the program encounters the error 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.

Test again to see what happens when you try to register someone with the same username and/or email address.

You will *never* be able to anticipate every error, however error handlers (when used right) help to catch common errors so that they are handled gracefully. This not only makes your application look professional, but helps to prevent any security vulnerabilities as well.

Unit Testing

Conducting unit tests on your application is an absolute necessity, especially as your app grows in size and complexity. Tests help ensure that as the complexity of our application grows the various moving parts continue to work together in a harmonious fashion. The writing and running of tests can be difficult at first, but once you understand the importance and practice, practice, practice, the process is simple.

Tests help reveal

1. When code isn't working,
2. When code breaks, and
3. Why you wrote the code in the first place, since they can help you gain understanding of your code base in general.

Every time you add a feature to our application, fix a bug, or change some code you should make sure the code is adequately covered by tests and that the tests all pass after you're done.

You also can define the tests before you write any code. This is called Test Driven, or Test First, Development. This helps you fully hash out your application's requirements. It also prevent over-coding: You develop your application until the unit 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.

Although there is an official Flask extension called Flask-Testing to perform tests, it's recommended to start with the pre-installed test package/framework that comes with Python, aptly named [unittest](#). It's not quite as easy to use, but you will learn more. Once you get comfortable with writing unit tests, then move on to the extension if you'd like.

Each test is written as a separate function within a larger class. 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 whether the desired outcome varies from the actual outcome.

All tests begin with this basic framework:

```
1 import unittest
2
```



```

3 class TestCase(unittest.TestCase):
4
5     # place your test functions here
6
7 if __name__ == '__main__':
8     unittest.main()

```

Let's create a base unit test script:

```

1 # project/test.py
2
3
4 import os
5 import unittest
6
7 from views import app, db
8 from _config import basedir
9 from models import User
10
11 TEST_DB = 'test.db'
12
13
14 class AllTests(unittest.TestCase):
15
16     #####
17     ##### setup and teardown #####
18     #####
19
20     # executed prior to each test
21     def setUp(self):
22         app.config['TESTING'] = True
23         app.config['WTF_CSRF_ENABLED'] = False
24         app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + \
25             os.path.join(basedir, TEST_DB)
26         self.app = app.test_client()
27         db.create_all()
28
29     # executed after each test
30     def tearDown(self):
31         db.session.remove()
32         db.drop_all()

```

```

33
34     # each test should start with 'test'
35     def test_user_setup(self):
36         new_user = User("michael", "michael@mherman.org",
37                         "michaelherman")
38         db.session.add(new_user)
39         db.session.commit()
40
41 if __name__ == "__main__":
42     unittest.main()

```

Save it as *test.py* in the root directory and run it:

```

1 $ python test.py
2 .
3 -----
4 Ran 1 test in 0.223s
5
6 OK

```

It passed!

What happened as we ran this script?

1. The `setUp` method() was invoked which created a test database (if it doesn't exist yet) and initialized the database schema from the main database (e.g., creates the tables, relationships, etc.). It also created a **test client**, which we can use to send requests to and then test the responses. It's essentially mocking out our entire application.
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?

Now, while the `tearDown()` method is still commented out, run the test script a second time.

```

1 sqlalchemy.exc.IntegrityError: (IntegrityError) UNIQUE constraint
  failed: users.email 'INSERT INTO users (name, email, password)
  VALUES (?, ?, ?)' ('michael', 'michael@mherman.org',
  'michaelherman')

```

```
2
3 -----
4 Ran 1 test in 0.044s
5
6 FAILED (errors=1)
```

What happened?

As you can see, we got an error this time because the name and email must be unique (as defined in our `User()` class in *models.py*).

Delete *test.db* 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:

```
1 # project/test.py
2
3
4 import os
5 import unittest
6
7 from views import app, db
8 from _config import basedir
9 from models import User
10
11 TEST_DB = 'test.db'
12
13
14 class AllTests(unittest.TestCase):
15
16     #####
17     ##### setup and teardown #####
18     #####
19
```

```

20     # executed prior to each test
21     def setUp(self):
22         app.config['TESTING'] = True
23         app.config['WTF_CSRF_ENABLED'] = False
24         app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + \
25             os.path.join(basedir, TEST_DB)
26         self.app = app.test_client()
27         db.create_all()
28
29     # executed after each test
30     def tearDown(self):
31         db.session.remove()
32         db.drop_all()
33
34     # each test should start with 'test'
35     def test_users_can_register(self):
36         new_user = User("michael", "michael@mherman.org",
37             "michaelherman")
38         db.session.add(new_user)
39         db.session.commit()
40         test = db.session.query(User).all()
41         for t in test:
42             t.name
43             assert t.name == "michael"
44
45 if __name__ == "__main__":
46     unittest.main()

```

In this example, we're testing the same thing: 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 this program in the current form. It should pass.

Then change the assert statement to -

```

1 assert t.name != "michael"

```

Now you can see what an assertion error looks like:

```

1 =====
2 FAIL: test_users_can_register (__main__.AllTests)

```

```

3 -----
4 Traceback (most recent call last):
5   File "test.py", line 41, in test_users_can_register
6     assert t.name != "michael"
7 AssertionError
8
9 -----
10 Ran 1 test in 0.044s
11
12 FAILED (failures=1)

```

Change the assert statement back to `assert t.name == "michael".`

Test again to make sure it works.

What are we testing?

Now think about this test for a minute. What exactly are we testing? Well, we're testing that the data was inserted correctly in the database. In essence, we're testing both SQLAlchemy and the database itself, not the code we wrote. As a general rule, you should avoid testing code for functionality that we have not explicitly written, since that code *should* already be tested. Also, since we are hitting a database, this is technically an integration test, not a unit test. Semantics aside, this is not a course on testing. Testing is an art - and one that takes years to hone. The goal here is to show you *how* to test, and then, over time, you'll learn *what* to test.

Let's test 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. Let's just get the tests working.

Alright. Start testing. GO! Any ideas where to begin? This is the hard part, which is why test driven development is so powerful, since when you test as you go in theory you know your code is working before moving on to the next piece of functionality. Unfortunately, test drive development is built on the notion that you are writing the right tests. Again, this takes time and a lot of practice.

Try this: Take out a piece of paper and go through your *views.py* file, writing everything down that should be tested - that is, every piece of functionality and/or logic we wrote. Go back and look at all the functions in your *views.py* file. You want *at least* enough tests to cover each one of those functions. Try to break this down between users and tasks. Once done, compare your notes with my table below.

Function	Not logged in	Logged in	What are we testing?
login()	X	X	Form is present on login page
login()	X	-	Un-registered users cannot login
login()	X	-	Registered users can login (form validation)
register()	X	-	Form is present on register page
register()	X	-	Users can register (form validation)
logout()	-	X	Users can logout
tasks()	-	X	Users can access tasks
tasks()	-	X	Users can add tasks (form validation)
tasks()	-	X	Users can complete tasks
tasks()	-	X	Users can delete tasks

How does your outline compare? 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.

Unit Tests for users

Form is present on login page

```

1 def test_form_is_present_on_login_page(self):
2     response = self.app.get('/')
3     self.assertEqual(response.status_code, 200)
4     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 the asserts that response 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? Well, we don't know what is returned in the case of a 200 responses. It could be JSON or an entirely different HTML page, for example.

Un-registered users cannot login

First, let's use a helper method to log users in as necessary. Since we'll have to do this multiple times it makes sense to abstract the code out into a separate method, and then call this method as necessary. This helps to keep our code DRY.

```
1 def login(self, name, password):
2     return self.app.post('/', data=dict(
3         name=name, password=password), follow_redirects=True)
```

The test should make sense:

```
1 def test_users_cannot_login_unless_registered(self):
2     response = self.login('foo', 'bar')
3     self.assertIn(b'Invalid username or password.', response.data)
```

Registered users can login (form validation)

So, here we're testing the form validation, which is essential that it's done right because we want our data to make sense. On the database level, since we are using SQLAlchemy, any data that contains special characters, that are generally used for SQL injection, are automatically escaped. So, as long as you are using 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, let's test the model.

Let's start with some valid data.

Add another helper method:

```
1 def register(self, name, email, password, confirm):
2     return self.app.post(
3         'register/',
4         data=dict(name=name, email=email, password=password,
5                 confirm=confirm),
6         follow_redirects=True)
```

This just registers a new user.

Now test that the user can login:

```
1 def test_users_can_login(self):
2     self.register('Michael', 'michael@realpython.com', 'python',
3                 'python')
```

```

3     response = self.login('Michael', 'python')
4     self.assertIn('Welcome!', response.data)

```

Finally, let's test some bad data and see how far the process gets

```

1 def test_invalid_form_data(self):
2     self.register('Michael', 'michael@realpython.com', 'python',
3                   'python')
4     response = self.login('alert("alert box!");', 'foo')
5     self.assertIn(b'Invalid username or password.', response.data)

```

This is a very, very similar test to `test_users_cannot_login_unless_registered()`. We'll leave it for now, but it may be something that we get rid of (or combine) in the future.

Form is present on register page

```

1 def test_form_is_present_on_register_page(self):
2     response = self.app.get('register/')
3     self.assertEqual(response.status_code, 200)
4     self.assertIn(b'Please register to access the task list.',
5                   response.data)

```

This should be straightforward. Perhaps 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)

```

1 def test_user_registration(self):
2     self.app.get('register/', follow_redirects=True)
3     response = self.register(
4         'Michael', 'michael@realpython.com', 'python', 'python')
5     self.assertIn(b'Thanks for registering. Please login.',
6                   response.data)

```

Look back at your tests. We're kind of already testing for this, right? Should we refactor?

```

1 def test_user_registration_error(self):
2     self.app.get('register/', follow_redirects=True)
3     self.register('Michael', 'michael@realpython.com', 'python',
4                   'python')
5     self.app.get('register/', follow_redirects=True)
6     response = self.register(

```



```

6         'Michael', 'michael@realpython.com', 'python', 'python'
7     )
8     self.assertIn(
9         b'That username and/or email already exist.',
10        response.data
11    )

```

This code works, but is it DRY?

Users can logout

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 methods:

```

1 def logout(self):
2     return self.app.get('/logout/', follow_redirects=True)

```

Now we can test logging out for both logged in and not logged in users:

```

1 def test_logged_in_users_can_logout(self):
2     self.register('Fletcher', 'fletcher@realpython.com',
3                 'python101', 'python101')
4     self.login('Fletcher', 'python101')
5     response = self.logout()
6     self.assertIn(b'Goodbye!', response.data)
7
8 def test_not_logged_in_users_cannot_logout(self):
9     response = self.logout()
10    self.assertNotIn(b'Goodbye!', response.data)

```

Run the tests. You should get a failure. Basically, not logged in users can still access that end point, /logout. Let's fix that. It's easy.

Simply add the `@login_required` decorator to the view:

```

1 @app.route('/logout/')
2 @login_required
3 def logout():
4     session.pop('logged_in', None)
5     session.pop('user_id', None)
6     flash('Goodbye!')
7     return redirect(url_for('login'))

```

Test it again. They should all pass.

Users can access tasks

Similar to the last test, but this time we're just checking a different endpoint.

```
1 def test_logged_in_users_can_access_tasks_page(self):
2     self.register(
3         'Fletcher', 'fletcher@realpython.com', 'python101',
4         'python101'
5     )
6     self.login('Fletcher', 'python101')
7     response = self.app.get('tasks/')
8     self.assertEqual(response.status_code, 200)
9     self.assertIn(b'Add a new task:', response.data)
10
11 def test_not_logged_in_users_cannot_access_tasks_page(self):
12     response = self.app.get('tasks/', follow_redirects=True)
13     self.assertIn(b'You need to login first.', response.data)
```

Seems like you could combine the last two tests?

Unit Tests for users

For these next set of tests, we'll assume that only users that are logged in can add, complete, or delete tasks since we already know that only logged in users can access the 'tasks/' endpoint? Sound reasonable?

Also, lets add two more helper methods:

```
1 def create_user(self, name, email, password):
2     new_user = User(name=name, email=email, password=password)
3     db.session.add(new_user)
4     db.session.commit()
5
6 def create_task(self):
7     return self.app.post('add/', data=dict(
8         name='Go to the bank',
9         due_date='02/05/2014',
10        priority='1',
11        posted_date='02/04/2014',
```

```

12         status='1'
13     ), follow_redirects=True)

```

Users can add tasks (form validation)

```

1 def test_users_can_add_tasks(self):
2     self.create_user('Michael', 'michael@realpython.com', 'python')
3     self.login('Michael', 'python')
4     self.app.get('tasks/', follow_redirects=True)
5     response = self.create_task()
6     self.assertIn(
7         b'New entry was successfully posted. Thanks.', response.data
8     )

```

What if there's an error?

```

1 def test_users_cannot_add_tasks_when_error(self):
2     self.create_user('Michael', 'michael@realpython.com', 'python')
3     self.login('Michael', 'python')
4     self.app.get('tasks/', follow_redirects=True)
5     response = self.app.post('add/', data=dict(
6         name='Go to the bank',
7         due_date='',
8         priority='1',
9         posted_date='02/05/2014',
10        status='1'
11    ), follow_redirects=True)
12    self.assertIn(b'This field is required.', response.data)

```

Users can complete tasks

```

1 def test_users_can_complete_tasks(self):
2     self.create_user('Michael', 'michael@realpython.com', 'python')
3     self.login('Michael', 'python')
4     self.app.get('tasks/', follow_redirects=True)
5     self.create_task()
6     response = self.app.get("complete/1/", follow_redirects=True)
7     self.assertIn(b'The task is complete. Nice.', response.data)

```

Users can delete tasks

```

1 def test_users_can_delete_tasks(self):
2     self.create_user('Michael', 'michael@realpython.com', 'python')
3     self.login('Michael', 'python')
4     self.app.get('tasks/', follow_redirects=True)
5     self.create_task()
6     response = self.app.get("delete/1/", follow_redirects=True)
7     self.assertIn(b'The task was deleted.', response.data)

```

Well, what did we miss? Remember when we were setting up the ‘users’ table and defining the relationship between users and tasks? Well, it should be 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*. So, if user A adds a task, only user A can update and/or delete that task. Let’s test that out.

```

1 def
2     test_users_cannot_complete_tasks_that_are_not_created_by_them(self):
3         self.create_user('Michael', 'michael@realpython.com', 'python')
4         self.login('Michael', 'python')
5         self.app.get('tasks/', follow_redirects=True)
6         self.create_task()
7         self.logout()
8         self.create_user('Fletcher', 'fletcher@realpython.com',
9                             'python101')
10        self.login('Fletcher', 'python101')
11        self.app.get('tasks/', follow_redirects=True)
12        response = self.app.get("complete/1/", follow_redirects=True)
13        self.assertNotIn(
14            b'The task is complete. Nice.', response.data
15        )

```

By now you probably already know that test is going to fail, just from manual testing alone. So, this is a perfect time to start the TDD process. That said, you deserve a break from *FlaskTaskr*. Take a look at the basics of HTML and CSS, in the next chapter.

Your project structure should now look like this:

```

1
2 _config.py
3 db_create.py
4 db_migrate.py
5 flasktaskr.db
6 forms.py

```

```
7 models.py
8 run.py
9 static
10     css
11         main.css
12     img
13     js
14 templates
15     _base.html
16     login.html
17     register.html
18     tasks.html
19 test.db
20 test.py
21 views.py
```

The code for this chapter can be found in the *flasktaskr-03* folder in the course [repository](#)

Chapter 12

Interlude: Introduction to HTML and CSS

Let's take a quick 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 this first part we will be addressing HTML and CSS.

NOTE: This is a beginner tutorial. If you already have a basic understanding of HTML and CSS, feel free to skip.

Websites 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 webpage 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 webpages, CSS is used to define the look and feel of a webpage.

Now let's turn to an actual web page ..

HTML

HTML gives a web pages structure, allowing you to view it from a web browser.

Start by adding some basic structure:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4   </head>
5   <body>
6   </body>
7 </html>
```

Copy and paste this basic webpage structure into your text editor. Save this file as *index.html* in a new directory called “front-end”.

This structure is commonly referred to as a boilerplate template. Such templates are used to speed up development so you don’t have to code the features common to every single webpage each time you create a new page. Most boilerplates include more features (or boilerplate code), but let’s start with the basics.

What’s going on?

1. The first line, `<!DOCTYPE html>` is the document type declaration, which tells the browser the version of HTML the page is using (HTML5, in our case). 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 we wish to use in our web page, 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.

Let’s add four tags:

1. title <title>
2. heading <h1>
3. break

4. paragraph <p>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>My Todo List</title>
5   </head>
6   <body>
7     <h1>My Todo List</h1>
8     <p>Get yourself organized!</p>
9     <br>
10  </body>
11 </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 (sometimes referred to as selectors) are key-value pairs used to select the tag for either applying Javascript or CSS styles. Selectors in most cases are either **ids** or **classes**.

Mozilla has an excellent reference guide [here](#) for all HTML elements.

Additional Tags

Let's add some more tags.


```
1 <!doctype html>
2 <html>
3   <head>
4     <title>My Todo List</title>
5   </head>
6   <body>
7
8     <h1>My Todo List</h1>
9     <p>Get yourself organized!</p>
10    <br>
11    <form>
12      <input type="text" placeholder="Enter a todo...">
13      <br>
14    </form>
15    <button>Submit!</button>
16    <br>
17
18  </body>
19 </html>
```

Essentially, 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, to put it nicely. Fortunately, we can quickly change that with CSS!

On to CSS ..

CSS

While HTML provides, structure, CSS is used for styling, making webpages look nice. 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 (or rules) 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. Or External. Inline styles are placed directly in the tag; these should be avoided, though, as it's best practice to keep HTML and CSS styles separated (don't mix structure with presentation!). Internal styles fall within the head of a website. Again, these should be avoided as well due to reasons mentioned before. Read more about this [here](#).

First, we need to "link" our HTML page and CSS stylesheet. To do so, add the following code to the <head> section of the HTML page just above the title:

```
1 <link rel="stylesheet" type="text/css"
  href="http://netdna.bootstrapcdn.com/
2   bootswatch/3.0.3/flatly/bootstrap.min.css">
3 <link rel="stylesheet" type="text/css" href="main.css">
```

Your code should now look like this:

```
1 <!doctype html>
2 <html>
3   <head>
4     <link rel="stylesheet" type="text/css"
      href="http://netdna.bootstrapcdn.com/
5       bootswatch/3.0.3/flatly/bootstrap.min.css">
6     <link rel="stylesheet" type="text/css" href="main.css">
7     <title>My Todo List</title>
8   </head>
9   <body>
10
11     <div class="container">
12       <h1>My Todo List</h1>
13       <p>Get yourself organized!</p>
14       <br>
```

```

15     <form>
16         <input type="text" placeholder="Enter a todo...">
17         <br>
18     </form>
19     <button>Submit!</button>
20     <br>
21 </div>
22
23 </body>
24 </html>

```

Save the file. We are using two stylesheets. The first 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 see [this](#) blog post.

Check the page out in your browser. See the difference? Yes, it's subtle - but the font is different along with the style of the inputs.

Notice how I also added a `<div>` tag with the class `"container"`. This is a [bootstrap](#) class.

Now create a `main.css` file and save it in the same folder as your `index.html` file.

Add the following CSS to the file:

```

1 .container {
2     max-width: 500px;
3     padding-top: 50px;
4 }

```

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 words, 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. <http://cssvalues.com/> is an excellent resource for finding the acceptable values given a CSS property.

Putting it all together

First, add the following selectors to the HTML:

```
1 <!doctype html>
2 <html>
3   <head>
4     <link rel="stylesheet" type="text/css"
5       href="http://netdna.bootstrapcdn.com/
6       bootswatch/3.0.3/flatly/bootstrap.min.css">
7     <link rel="stylesheet" type="text/css" href="main.css">
8     <title>My Todo List</title>
9   </head>
10  <body>
11    <div class="container">
12      <h1>My Todo List</h1>
13      <p class="lead">Get yourself organized!</p>
14      <br>
15      <form id="my-form" role="form">
16        <input id="my-input" class="form-control" type="text"
17          placeholder="Enter a todo...">
18        <br>
19      </form>
20      <button class="btn btn-primary btn-md">Submit!</button>
21      <br>
22    </div>
23  </body>
24 </html>
```

NOTE: Do you see the selectors? Look for the new **ids** and **classes**. The **ids** will all be used for Javascript, while the **classes** are all bootstrap styles. If you're curious, check out the bootstrap [website](#) to see more info about these styles.

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 of time, plus you can make edits to any website, not just your own.

Open up the HTML page we worked on. Right Click on the heading. Select “Inspect 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:

```
1 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 and select “Edit as HTML” 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 find bugs and/or learn how to imitate a desired HTML, CSS, 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.

Back to Flask ...

Homework

- If you want some extra practice, go through the Codecademy series on [HTML and CSS](#).
- Want even more practice, try [this](#) fun game for learning CSS selectors.

Chapter 13

Flask: FlaskTaskr, Part 4 - Styles, Test Coverage, and Permissions

Back to *FlaskTaskr*. Fire up your virtualenv and then run the tests:

```
1 .....F.
2 =====
3 FAIL: test_users_cannot_complete_tasks_that_are_not_created_by_them
   (__main__.AllTests)
4 -----
5
6
7 -----
8 Ran 17 tests in 0.661s
9
10 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

Task	Complete
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:

```
1
2 static
3     css
4         bootstrap.min.css
5     js
6         jquery-1.11.3.min.js
7         bootstrap.min.js
8     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*:

```
1 <!-- styles -->
2 <link
    href="//maxcdn.bootstrapcdn.com/bootswatch/3.3.4/yeti/bootstrap.min.css"
    rel="stylesheet">
```

```

3
4 <!-- scripts -->
5 <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
6 <script
    src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js"></script>

```

Now add some bootstrap styles. Update your template so that it now looks like this:

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6     <meta name="viewport" content="width=device-width,
7       initial-scale=1">
8     <meta name="description" content="">
9     <meta name="author" content="">
10    <title>Welcome to FlaskTaskr!!!</title>
11    <!-- styles -->
12    <link
13      href="//maxcdn.bootstrapcdn.com/bootswatch/3.3.4/yeti/bootstrap.min.css"
14      rel="stylesheet">
15    <link rel="stylesheet" href="{{ url_for('static',
16      filename='css/main.css') }}">
17  </head>
18  <body>
19
20    <div class="navbar navbar-inverse navbar-fixed-top"
21      role="navigation">
22      <div class="container">
23        <div class="navbar-header">
24          <button type="button" class="navbar-toggle"
25            data-toggle="collapse" data-target=".navbar-collapse">
26            <span class="sr-only">Toggle navigation</span>
27            <span class="icon-bar"></span>
28            <span class="icon-bar"></span>
29            <span class="icon-bar"></span>
30          </button>
31          <a class="navbar-brand" href="/">FlaskTaskr</a>
32        </div>
33        <div class="collapse navbar-collapse">

```

```

28         <ul class="nav navbar-nav">
29             {% if not session.logged_in %}
30                 <li><a href="/register">Signup</a></li>
31             {% else %}
32                 <li><a href="/logout">Signout</a></li>
33             {% endif %}
34         </ul>
35     </div><!--/.nav-collapse -->
36 </div>
37 </div>
38
39 <div class="container">
40     <div class="content">
41
42         {% for message in get_flashed_messages() %}
43             <div class="alert alert-success">{{ message }}</div>
44         {% endfor %}
45
46         {% if error %}
47             <div class="error"><strong>Error:</strong> {{ error
48                 }}</div>
49         {% endif %}
50
51         {% block content %}
52
53     </div>
54     <div class="footer">
55         <hr>
56         <p>&copy; <a href="https://www.realpython.com">Real
57             Python</a></p>
58     </div><!-- /.container -->
59
60 <!-- scripts -->
61 <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
62 <script
63     src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.4/js/bootstrap.min.js"></script>
64 </body>
65 </html>

```

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:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Welcome to FlaskTaskr!!</title>
5     <link rel="stylesheet" href="{{ url_for('static',
6       filename='css/main.css') }}">
7   </head>
8   <body>
9     <div class="page">
10
11       {% for message in get_flashed_messages() %}
12         <div class="flash">{{ message }}</div>
13       {% endfor %}
14
15       {% if error %}
16         <div class="error"><strong>Error:</strong> {{ error }}</div>
17       {% endif %}
18
19       <br>
20
21       {% block content %}
22
23     </div>
24   </body>
25 </html>
```

Take a look at your app. See the difference. Now, let's update the child templates:

login.html

```
1 {% extends "_base.html" %}
2
3 {% block content %}
4
```

```

5 <h1>Welcome to FlaskTaskr.</h1>
6 <h3>Please sign in to access your task list</h3>
7
8 <form class="form-signin" role="form" method="post" action="/">
9   {{ form.csrf_token }}
10   <p>
11     {{ form.name(placeholder="email address") }}
12     <span class="error">
13       {% if form.name.errors %}
14         {% for error in form.name.errors %}
15           {{ error }}
16         {% endfor %}
17       {% endif %}
18     </span>
19   </p>
20   <p>
21     {{ form.password(placeholder="password") }}
22     <span class="error">
23       {% if form.password.errors %}
24         {% for error in form.password.errors %}
25           {{ error }}
26         {% endfor %}
27       {% endif %}
28     </span>
29   </p>
30   <button class="btn btn-sm btn-success" type="submit">Sign
31     in</button>
32   <br>
33   <p><em>Need an account? </em><a href="/register">Signup!!</a></p>
34 </form>
35
36 {% endblock %}

```

register.html

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr.</h1>

```

```

5 <h3>Please register to access the task list.</h3>
6 <form method="POST" action="/register/">
7   {{ form.csrf_token }}
8   <p>
9     {{ form.name(placeholder="name") }}
10    {% if form.name.errors %}
11      <span class="error">
12        {% for error in form.name.errors %}
13          {{ error }}
14        {% endfor %}
15      </span>
16    {% endif %}
17  </p>
18  <p>
19    {{ form.email(placeholder="email address") }}
20    {% if form.email.errors %}
21      <span class="error">
22        {% for error in form.email.errors %}
23          {{ error }}
24        {% endfor %}
25      </span>
26    {% endif %}
27  </p>
28  <p>
29    {{ form.password(placeholder="password") }}
30    {% if form.password.errors %}
31      <span class="error">
32        {% for error in form.password.errors %}
33          {{ error }}
34        {% endfor %}
35      </span>
36    {% endif %}
37  </p>
38  <p>
39    {{ form.confirm(placeholder="confirm password") }}
40    {% if form.confirm.errors %}
41      <span class="error">
42        {% for error in form.confirm.errors %}
43          {{ error }}
44        {% endfor %}

```

```

45     </span>
46     {% endif %}
47 </p>
48 <button class="btn btn-sm btn-success"
49     type="submit">Register</button>
50 <br>
51 <p><em>Already registered?</em> Click <a href="/">here</a> to
52     login.</p>
53 </form>
54 {% endblock %}

```

tasks.html

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr</h1>
5 <br>
6 <a href="/logout">Logout</a>
7 <div class="add-task">
8     <h3>Add a new task:</h3>
9     <form action="{{ url_for('new_task') }}" method="post">
10         {{ form.csrf_token }}
11         <p>
12             {{ form.name(placeholder="name") }}
13             <span class="error">
14                 {% if form.name.errors %}
15                     {% for error in form.name.errors %}
16                         {{ error }}
17                     {% endfor %}
18                 {% endif %}
19             </span>
20         </p>
21         <p>
22             {{ form.due_date(placeholder="due date") }}
23             <span class="error">
24                 {% if form.due_date.errors %}
25                     {% for error in form.due_date.errors %}
26                         {{ error }}

```

```

27         {% endfor %}
28     {% endif %}
29 </span>
30 </p>
31 <p>
32     {{ form.priority.label }}
33     {{ form.priority }}
34     <span class="error">
35         {% if form.priority.errors %}
36             {% for error in form.priority.errors %}
37                 {{ error }}
38             {% endfor %}
39         {% endif %}
40     </span>
41 </p>
42 <p><input class="btn btn-default" type="submit"
43     value="Submit"></p>
44 </form>
45 </div>
46 <div class="entries">
47     <br>
48     <br>
49     <h2>Open tasks:</h2>
50     <div class="datagrid">
51         <table>
52             <thead>
53                 <tr>
54                     <th width="200px"><strong>Task Name</strong></th>
55                     <th width="75px"><strong>Due Date</strong></th>
56                     <th width="100px"><strong>Posted Date</strong></th>
57                     <th width="50px"><strong>Priority</strong></th>
58                     <th width="90px"><strong>Posted By</strong></th>
59                     <th><strong>Actions</strong></th>
60                 </tr>
61             </thead>
62             {% for task in open_tasks %}
63                 <tr>
64                     <td width="200px">{{ task.name }}</td>
65                     <td width="75px">{{ task.due_date }}</td>
66                     <td width="100px">{{ task.posted_date }}</td>

```



```

66         <td width="50px">{{ task.priority }}</td>
67         <td width="90px">{{ task.poster.name }}</td>
68         <td>
69             <a href="{{ url_for('delete_entry', task_id =
70                 task.task_id) }}">Delete</a> -
71             <a href="{{ url_for('complete', task_id = task.task_id)
72                 }}">Mark as Complete</a>
73         </td>
74     </tr>
75     {% endfor %}
76 </table>
77 </div>
78 <br>
79 <div class="entries">
80     <h2>Closed tasks:</h2>
81     <div class="datagrid">
82         <table>
83             <thead>
84                 <tr>
85                     <th width="200px"><strong>Task Name</strong></th>
86                     <th width="75px"><strong>Due Date</strong></th>
87                     <th width="100px"><strong>Posted Date</strong></th>
88                     <th width="50px"><strong>Priority</strong></th>
89                     <th width="90px"><strong>Posted By</strong></th>
90                     <th><strong>Actions</strong></th>
91                 </tr>
92             </thead>
93             {% for task in closed_tasks %}
94                 <tr>
95                     <td width="200px">{{ task.name }}</td>
96                     <td width="75px">{{ task.due_date }}</td>
97                     <td width="100px">{{ task.posted_date }}</td>
98                     <td width="50px">{{ task.priority }}</td>
99                     <td width="90px">{{ task.poster.name }}</td>
100                     <td>
101                         <a href="{{ url_for('delete_entry', task_id =
102                             task.task_id) }}">Delete</a>

```

```

103         </tr>
104     {% endfor %}
105 </table>
106 </div>
107 </div>
108
109 {% endblock %}

```

Custom Styles

Update the styles in *main.css*:

```

1 body {
2     padding: 60px 0;
3     background: #ffffff;
4 }
5
6 .footer {
7     padding-top: 30px;
8 }
9
10 .error {
11     color: red;
12     font-size: .8em;
13 }

```

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:

```

1 $ python 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](#).

Install:

```
1 $ pip install coverage==3.7.1
2 $ pip freeze > requirements.txt
```

To run coverage, use the following command:

```
1 $ coverage run test.py
```

To get a command line coverage report run:

```
1 $ coverage report --omit=./env/*
2
3 Name                Stmts    Miss  Cover
4 -----
5 _config                7         0   100%
6 forms                 17         0   100%
7 models                33         2    94%
8 test                 120         0   100%
9 views                 89         4    96%
10 -----
11 TOTAL                 266         6    98%
```

NOTE: `--omit=./env/*` excludes all files within the virtualenv

To print a more robust HTML report:

```
1 $ 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:

```
1 else:
2     error = 'Both fields are required.'
```

Re-run coverage:

```
1 $ coverage run test.py
2 $ coverage html --omit=./env/*
```

Our coverage for the views should have jumped from 96% to 98%. Nice.

NOTE: Make sure to add both `.coverage` and “htmlcov” to the `.gitignore` file. These should not be part of your repository.

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:

```
1 $ pip install nose==1.3.4
2 $ pip freeze > requirements.txt
```

Running Nose

Now you should be able to run Nose, along with coverage:

```
1 $ 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==app` specifies which portion of the code to analyze.

Run the tests:

```
1 Name           Stmt  Miss  Cover
2 -----
3 _config         7      0  100%
4 forms          17      0  100%
5 models         33      2   94%
6 test           120      0  100%
7 views          88      3   97%
8 -----
9 TOTAL          265      5   98%
10 -----
11 Ran 17 tests in 2.904s
12
13 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 from the repository to ensure that we're on the same page.

Users can only update tasks that they created

Start by running the tests:

```
1 $ nosetests
```

As expected, we have one failure:

```
1 $ nosetests
2 .....F.....
3 =====
4 FAIL: test_users_cannot_complete_tasks_that_are_not_created_by_them
      (test_tasks.TasksTests)
5 -----
6
7
8 -----
```

```
9 Ran 21 tests in 1.029s
10
11 FAILED (failures=1)
```

Update the code

```
1 @app.route('/complete/<int:task_id>/')
2 @login_required
3 def complete(task_id):
4     new_id = task_id
5     task = db.session.query(Task).filter_by(task_id=new_id)
6     if session['user_id'] == task.first().user_id:
7         task.update({"status": "0"})
8         db.session.commit()
9         flash('The task is complete. Nice.')
10        return redirect(url_for('tasks'))
11    else:
12        flash('You can only update tasks that belong to you.')
13        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

```
1 $ nosetests
2 .....
3 -----
4 Ran 21 tests in 0.942s
5
6 OK
```

Yay!

Refactor

Before declaring a victory, let's update the test to check for the flashed message:

```
1 def
2     test_users_cannot_complete_tasks_that_are_not_created_by_them(self):
3         self.create_user('Michael', 'michael@realpython.com', 'python')
4         self.login('Michael', 'python')
```



```

4     self.app.get('tasks/', follow_redirects=True)
5     self.create_task()
6     self.logout()
7     self.create_user('Fletcher', 'fletcher@realpython.com',
8                       'python101')
9     self.login('Fletcher', 'python101')
10    self.app.get('tasks/', follow_redirects=True)
11    response = self.app.get("complete/1/", follow_redirects=True)
12    self.assertNotIn(
13        b'The task is complete. Nice.', response.data
14    )
15    self.assertIn(
16        b'You can only update tasks that belong to you.',
17        response.data
18    )

```

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

```

1  def
2      test_users_cannot_delete_tasks_that_are_not_created_by_them(self):
3      self.create_user('Michael', 'michael@realpython.com', 'python')
4      self.login('Michael', 'python')
5      self.app.get('tasks/', follow_redirects=True)
6      self.create_task()
7      self.logout()
8      self.create_user('Fletcher', 'fletcher@realpython.com',
9                        'python101')
10     self.login('Fletcher', 'python101')
11     self.app.get('tasks/', follow_redirects=True)
12     response = self.app.get("delete/1/", follow_redirects=True)
13     self.assertIn(
14         b'You can only delete tasks that belong to you.',
15         response.data
16     )

```

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.

```
1 @app.route('/delete/<int:task_id>/')
2 @login_required
3 def delete_entry(task_id):
4     new_id = task_id
5     task = db.session.query(Task).filter_by(task_id=new_id)
6     if session['user_id'] == task.first().user_id:
7         task.delete()
8         db.session.commit()
9         flash('The task was deleted. Why not add a new one?')
10        return redirect(url_for('tasks'))
11    else:
12        flash('You can only delete tasks that belong to you.')
13        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:

```

1 def test_default_user_role(self):
2
3     db.session.add(
4         User(
5             "Johnny",
6             "john@doe.com",
7             "johnny"
8         )
9     )
10
11     db.session.commit()
12
13     users = db.session.query(User).all()
14     print users
15     for user in users:
16         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:

```

1 class User(db.Model):
2
3     __tablename__ = 'users'
4
5     id = db.Column(db.Integer, primary_key=True)
6     name = db.Column(db.String, unique=True, nullable=False)
7     email = db.Column(db.String, unique=True, nullable=False)
8     password = db.Column(db.String, nullable=False)
9     tasks = db.relationship('Task', backref='poster')
10    role = db.Column(db.String, default='user')
11
12    def __init__(self, name=None, email=None, password=None,
13                  role=None):

```

```

13         self.name = name
14         self.email = email
15         self.password = password
16         self.role = role
17
18     def __repr__(self):
19         return '<User {0}>'.format(self.name)

```

Update the migration script:

```

1  # project/db_migrate.py
2
3
4  from views import db
5  # from datetime import datetime
6  from _config import DATABASE_PATH
7  import sqlite3
8
9  # with sqlite3.connect(DATABASE_PATH) as connection:
10
11     # get a cursor object used to execute SQL commands
12     c = connection.cursor()
13
14     # temporarily change the name of tasks table
15     c.execute("""ALTER TABLE tasks RENAME TO old_tasks""")
16
17     # recreate a new tasks table with updated schema
18     db.create_all()
19
20     # retrieve data from old_tasks table
21     c.execute("""SELECT name, due_date, priority,
22                 status FROM old_tasks ORDER BY task_id ASC""")
23
24     # save all rows as a list of tuples; set posted_date to now
    and user_id to 1
25     data = [(row[0], row[1], row[2], row[3],
26             datetime.now(), 1) for row in c.fetchall()]
27
28     # insert data to tasks table
29     c.executemany("""INSERT INTO tasks (name, due_date, priority,
    status,

```

```

30 #             posted_date, user_id) VALUES (?, ?, ?, ?, ?,
    ?)""" , data)
31
32 #         # delete old_tasks table
33 #         c.execute("DROP TABLE old_tasks")
34
35
36 with sqlite3.connect(DATABASE_PATH) as connection:
37
38     # get a cursor object used to execute SQL commands
39     c = connection.cursor()
40
41     # temporarily change the name of users table
42     c.execute("""ALTER TABLE users RENAME TO old_users""")
43
44     # recreate a new users table with updated schema
45     db.create_all()
46
47     # retrieve data from old_users table
48     c.execute("""SELECT name, email, password
49                 FROM old_users
50                 ORDER BY id ASC""")
51
52     # save all rows as a list of tuples; set role to 'user'
53     data = [(row[0], row[1], row[2],
54              'user') for row in c.fetchall()]
55
56     # insert data to users table
57     c.executemany("""INSERT INTO users (name, email, password,
58                                     role) VALUES (?, ?, ?, ?)""" , data)
59
60     # delete old_users table
61     c.execute("DROP TABLE old_users")

```

Run the migration:

```

1 $ 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_task.py`:

```
1 def
2     test_admin_users_can_complete_tasks_that_are_not_created_by_them(self):
3         self.create_user()
4         self.login('Michael', 'python')
5         self.app.get('tasks/', follow_redirects=True)
6         self.create_task()
7         self.logout()
8         self.create_admin_user()
9         self.login('Superman', 'allpowerful')
10        self.app.get('tasks/', follow_redirects=True)
11        response = self.app.get("complete/1/", follow_redirects=True)
12        self.assertNotIn(
13            'You can only update tasks that belong to you.',
14            response.data
15        )
16
17 def
18     test_admin_users_can_delete_tasks_that_are_not_created_by_them(self):
19         self.create_user()
20         self.login('Michael', 'python')
21         self.app.get('tasks/', follow_redirects=True)
22         self.create_task()
23         self.logout()
24         self.create_admin_user()
25         self.login('Superman', 'allpowerful')
26         self.app.get('tasks/', follow_redirects=True)
27         response = self.app.get("delete/1/", follow_redirects=True)
28         self.assertNotIn(
29             'You can only delete tasks that belong to you.',
30             response.data
31         )
```

Then add a new helper method:

```

1 def create_admin_user(self):
2     new_user = User(
3         name='Superman',
4         email='admin@realpython.com',
5         password='allpowerful',
6         role='admin'
7     )
8     db.session.add(new_user)
9     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:

```

1 @app.route('/logout/')
2 @login_required
3 def logout():
4     session.pop('logged_in', None)
5     session.pop('user_id', None)
6     session.pop('role', None)
7     flash('Goodbye!')
8     return redirect(url_for('login'))
9
10
11 @app.route('/', methods=['GET', 'POST'])
12 def login():
13     error = None
14     form = LoginForm(request.form)
15     if request.method == 'POST':
16         if form.validate_on_submit():
17             user =
18                 User.query.filter_by(name=request.form['name']).first()
19             if user is not None and user.password ==
20                 request.form['password']:
21                 session['logged_in'] = True
22                 session['user_id'] = user.id
23                 session['role'] = user.role

```

```

22         flash('Welcome!')
23         return redirect(url_for('tasks'))
24     else:
25         error = 'Invalid username or password.'
26     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:

```

1 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:

```

1 $ nosetests
2 .....
3 -----
4 Ran 25 tests in 1.252s
5
6 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!

Chapter 14

Flask: FlaskTaskr, Part 5 - Blueprints

Last time we updated the styles, updated our test workflow by adding Coverage and Nose, and added permissions:

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	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
Building a REST API	No
Boilerplate and Workflow	No
Continuous Integration and Delivery	No

As promised, let's convert our app to utilize [Blueprints](#). Before we start, be sure to-

1. Read about the benefits of using Blueprints from the official Flask [documentation](#).
2. Run your test suite one more time, making sure all tests pass.
3. Make sure all your project's dependencies are logged in *requirements.txt*: `pip freeze > requirements.txt`.
4. Commit your code to your local repo, just in case you make a *huge* mistake and want to revert back.

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 stacking/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 from the blog example. Perhaps your directory structure looks like this, where `admin`, `posts`, and `users` represent separate apps.

```
1
2 _config.py
3 run.py
4 blog
5     __init__.py
6     admin
7         static
8         templates
9         views.py
10    models.py
11    posts
12        static
13        templates
14        views.py
15    users
16        static
17        templates
18        views.py
```

Let's look at example of what the 'posts' Blueprint would look like.

```
1 # blog/posts/views.py
2
3
4 from flask import Blueprint, render_template, url_for
5
6 from _config import db
7 from forms import PostForm
8 from models import Post
9
10 # create a blueprint object
11 blueprint = Blueprint('blog_posts', __name__,)
12
13
14 @blueprint.route("/")
```

```

15 def read_posts():
16     posts = Post.query.all()
17     return render_template('posts.html', posts=posts)
18
19 @blueprint.route("/add/", methods=['GET', 'POST'])
20 def create_posts():
21     form = PostForm()
22     if form.validate_on_submit():
23         new_post = Post(form.title.data, form.description.data)
24         db.session.add(new_post)
25         db.session.commit()
26         flash('Post added!')
27         return redirect(url_for('read_posts'))
28     return render_template('blog/add.html', form=form)

```

Here, we initialized a new Blueprint, assigning it to the blueprint variable. From there, we bound each function to the `@blueprint.route` decorator. This let's Flask know that each function is available for use within the entire application.

Again, the above code defined the Blueprint, now we need to register it against the Flask object:

```

1 # blog/init.py
2
3 from flask import Flask
4 from posts.views import blueprint
5
6 blueprint = Flask(__name__)
7 blueprint.register_blueprint(blueprint)

```

That's it.

Refactoring our app

Now, let's convert our app, *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 individual function:

1. **Users:** handles user login/logout, registration, and authentication
2. **Tasks:** handles the CRUD functionality associated with our tasks

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:

```
1  __init__.py
2  forms.py
3  static
4  templates
5  views.py
```

Our current structure looks like this:

```
1  _config.py
2  db_create.py
3  db_migrate.py
4  flasktaskr.db
5  forms.py
6  models.py
7  run.py
8  static
9      css
10         main.css
11         img
12         js
13     templates
14         _base.html
```

```
16     login.html
17     register.html
18     tasks.html
19 test.db
20 test_tasks.py
21 test_users.py
22 views.py
```

After breaking our app up using Blueprints (don't do this yet), our application will have the following structure:

```
1
2 db_create.py
3 db_migrate.py
4 project
5     __init__.py
6     _config.py
7     flasktaskr.db
8     models.py
9     static
10         css
11             main.css
12         img
13         js
14     tasks
15         __init__.py
16         forms.py
17         views.py
18     templates
19         _base.html
20         login.html
21         register.html
22         tasks.html
23 test.db
24 users
25     __init__.py
26     forms.py
27     views.py
28 requirements.txt
29 run.py
30 tests
```

```
31     test_tasks.py
32     test_users.py
```

Study this new structure. Take note of the difference between the files and folders within each of the Blueprints, ‘users’ and ‘tasks’, and the project root.

Let’s walk through the conversion process.

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” 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:

```
1
2 db_create.py
3 db_migrate.py
4 project
5     __init__.py
6     _config.py
7     forms.py
8     models.py
9     static
10         css
11             main.css
12         img
13         js
```



```

14     tasks
15         __init__.py
16     templates
17         _base.html
18         login.html
19         register.html
20         tasks.html
21     users
22         __init__.py
23     views.py
24 run.py
25 tests
26     test_tasks.py
27     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*. 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

```

1  # project/users/views.py
2
3
4  #####
5  ##### imports #####
6  #####
7
8  from functools import wraps
9  from flask import flash, redirect, render_template, \
10     request, session, url_for, Blueprint
11  from sqlalchemy.exc import IntegrityError
12
13  from .forms import RegisterForm, LoginForm
14  from project import db

```

```

15 from project.models import User
16
17
18 #####
19 ##### config #####
20 #####
21
22 users_blueprint = Blueprint('users', __name__)
23
24
25 #####
26 ##### helper functions #####
27 #####
28
29
30 def login_required(test):
31     @wraps(test)
32     def wrap(*args, **kwargs):
33         if 'logged_in' in session:
34             return test(*args, **kwargs)
35         else:
36             flash('You need to login first.')
37             return redirect(url_for('users.login'))
38     return wrap
39
40
41 #####
42 ##### routes #####
43 #####
44
45 @users_blueprint.route('/logout/')
46 @login_required
47 def logout():
48     session.pop('logged_in', None)
49     session.pop('user_id', None)
50     session.pop('role', None)
51     flash('Goodbye!')
52     return redirect(url_for('users.login'))
53
54

```

```

55 @users_blueprint.route('/', methods=['GET', 'POST'])
56 def login():
57     error = None
58     form = LoginForm(request.form)
59     if request.method == 'POST':
60         if form.validate_on_submit():
61             user =
62                 User.query.filter_by(name=request.form['name']).first()
63             if user is not None and user.password ==
64                 request.form['password']:
65                 session['logged_in'] = True
66                 session['user_id'] = user.id
67                 session['role'] = user.role
68                 flash('Welcome!')
69                 return redirect(url_for('tasks.tasks'))
70             else:
71                 error = 'Invalid username or password.'
72         return render_template('login.html', form=form, error=error)
73
74 @users_blueprint.route('/register/', methods=['GET', 'POST'])
75 def register():
76     error = None
77     form = RegisterForm(request.form)
78     if request.method == 'POST':
79         if form.validate_on_submit():
80             new_user = User(
81                 form.name.data,
82                 form.email.data,
83                 form.password.data,
84             )
85             try:
86                 db.session.add(new_user)
87                 db.session.commit()
88                 flash('Thanks for registering. Please login.')
89                 return redirect(url_for('users.login'))
90             except IntegrityError:
91                 error = 'That username and/or email already exist.'
92                 return render_template('register.html', form=form,
93                                     error=error)

```

```
92     return render_template('register.html', form=form, error=error)
```

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

```
1  # project/users/forms.py
2
3
4  from flask_wtf import Form
5  from wtforms import StringField, PasswordField
6  from wtforms.validators import DataRequired, Length, EqualTo, Email
7
8
9  class RegisterForm(Form):
10     name = StringField(
11         'Username',
12         validators=[DataRequired(), Length(min=6, max=25)]
13     )
14     email = StringField(
15         'Email',
16         validators=[DataRequired(), Email(), Length(min=6, max=40)]
17     )
18     password = PasswordField(
19         'Password',
20         validators=[DataRequired(), Length(min=6, max=40)])
21     confirm = PasswordField(
22         'Repeat Password',
23         validators=[DataRequired(), EqualTo('password')]
24     )
25
26
27  class LoginForm(Form):
28     name = StringField(
29         'Username',
30         validators=[DataRequired()]
31     )
32     password = PasswordField(
33         'Password',
```

```

34     validators=[DataRequired()]
35 )

```

Tasks Blueprint

Views

```

1  # project/tasks/views.py
2
3
4  import datetime
5  from functools import wraps
6  from flask import flash, redirect, render_template, \
7     request, session, url_for, Blueprint
8
9  from .forms import AddTaskForm
10 from project import db
11 from project.models import Task
12
13
14 #####
15 ##### config #####
16 #####
17
18 tasks_blueprint = Blueprint('tasks', __name__)
19
20
21 #####
22 ##### helper functions #####
23 #####
24
25 def login_required(test):
26     @wraps(test)
27     def wrap(*args, **kwargs):
28         if 'logged_in' in session:
29             return test(*args, **kwargs)
30         else:
31             flash('You need to login first.')
32             return redirect(url_for('users.login'))
33     return wrap

```

```

34
35
36 def open_tasks():
37     return db.session.query(Task).filter_by(
38         status='1').order_by(Task.due_date.asc())
39
40
41 def closed_tasks():
42     return db.session.query(Task).filter_by(
43         status='0').order_by(Task.due_date.asc())
44
45
46 #####
47 #### routes ####
48 #####
49
50 @tasks_blueprint.route('/tasks/')
51 @login_required
52 def tasks():
53     return render_template(
54         'tasks.html',
55         form=AddTaskForm(request.form),
56         open_tasks=open_tasks(),
57         closed_tasks=closed_tasks()
58     )
59
60
61 @tasks_blueprint.route('/add/', methods=['GET', 'POST'])
62 @login_required
63 def new_task():
64     error = None
65     form = AddTaskForm(request.form)
66     if request.method == 'POST':
67         if form.validate_on_submit():
68             new_task = Task(
69                 form.name.data,
70                 form.due_date.data,
71                 form.priority.data,
72                 datetime.datetime.utcnow(),
73                 '1',

```

```

74         session['user_id']
75     )
76     db.session.add(new_task)
77     db.session.commit()
78     flash('New entry was successfully posted. Thanks.')
79     return redirect(url_for('tasks.tasks'))
80 return render_template(
81     'tasks.html',
82     form=form,
83     error=error,
84     open_tasks=open_tasks(),
85     closed_tasks=closed_tasks()
86 )
87
88
89 @tasks_blueprint.route('/complete/<int:task_id>/')
90 @login_required
91 def complete(task_id):
92     new_id = task_id
93     task = db.session.query(Task).filter_by(task_id=new_id)
94     if session['user_id'] == task.first().user_id or
95         session['role'] == "admin":
96         task.update({"status": "0"})
97         db.session.commit()
98         flash('The task is complete. Nice.')
99         return redirect(url_for('tasks.tasks'))
100     else:
101         flash('You can only update tasks that belong to you.')
102         return redirect(url_for('tasks.tasks'))
103
104 @tasks_blueprint.route('/delete/<int:task_id>/')
105 @login_required
106 def delete_entry(task_id):
107     new_id = task_id
108     task = db.session.query(Task).filter_by(task_id=new_id)
109     if session['user_id'] == task.first().user_id or
110         session['role'] == "admin":
111         task.delete()
112         db.session.commit()

```

```

112     flash('The task was deleted. Why not add a new one?')
113     return redirect(url_for('tasks.tasks'))
114 else:
115     flash('You can only delete tasks that belong to you.')
116     return redirect(url_for('tasks.tasks'))

```

Forms

```

1 # project/tasks/forms.py
2
3
4 from flask_wtf import Form
5 from wtforms import StringField, DateField, IntegerField, \
6     SelectField
7 from wtforms.validators import DataRequired
8
9
10 class AddTaskForm(Form):
11     task_id = IntegerField()
12     name = StringField('Task Name', validators=[DataRequired()])
13     due_date = DateField(
14         'Date Due (mm/dd/yyyy)',
15         validators=[DataRequired()], format='%m/%d/%Y'
16     )
17     priority = SelectField(
18         'Priority',
19         validators=[DataRequired()],
20         choices=[
21             ('1', '1'), ('2', '2'), ('3', '3'), ('4', '4'), ('5',
22                 '5'),
23             ('6', '6'), ('7', '7'), ('8', '8'), ('9', '9'), ('10',
24                 '10')
25         ]
26     )
27     status = IntegerField('Status')

```

tasks.html:

```

1 {% extends "_base.html" %}
2 {% block content %}
3
4 <h1>Welcome to FlaskTaskr</h1>

```



```

5 <br>
6 <a href="/logout">Logout</a>
7 <div class="add-task">
8   <h3>Add a new task:</h3>
9   <form action="{% url_for('tasks.new_task') %}" method="post">
10     {{ form.csrf_token }}
11     <p>
12       {{ form.name(placeholder="name") }}
13       <span class="error">
14         {% if form.name.errors %}
15           {% for error in form.name.errors %}
16             {{ error }}
17           {% endfor %}
18         {% endif %}
19       </span>
20     </p>
21     <p>
22       {{ form.due_date(placeholder="due date") }}
23       <span class="error">
24         {% if form.due_date.errors %}
25           {% for error in form.due_date.errors %}
26             {{ error }}
27           {% endfor %}
28         {% endif %}
29       </span>
30     </p>
31     <p>
32       {{ form.priority.label }}
33       {{ form.priority }}
34       <span class="error">
35         {% if form.priority.errors %}
36           {% for error in form.priority.errors %}
37             {{ error }}
38           {% endfor %}
39         {% endif %}
40       </span>
41     </p>
42     <p><input class="btn btn-default" type="submit"
43       value="Submit"></p>
44   </form>

```

```

44 </div>
45 <div class="entries">
46   <br>
47   <br>
48   <h2>Open tasks:</h2>
49   <div class="datagrid">
50     <table>
51       <thead>
52         <tr>
53           <th width="200px"><strong>Task Name</strong></th>
54           <th width="75px"><strong>Due Date</strong></th>
55           <th width="100px"><strong>Posted Date</strong></th>
56           <th width="50px"><strong>Priority</strong></th>
57           <th width="90px"><strong>Posted By</strong></th>
58           <th><strong>Actions</strong></th>
59         </tr>
60       </thead>
61       {% for task in open_tasks %}
62         <tr>
63           <td width="200px">{{ task.name }}</td>
64           <td width="75px">{{ task.due_date }}</td>
65           <td width="100px">{{ task.posted_date }}</td>
66           <td width="50px">{{ task.priority }}</td>
67           <td width="90px">{{ task.poster.name }}</td>
68           <td>
69             <a href="{% url_for('tasks.delete_entry', task_id =
              task.task_id) %}">Delete</a> -
70             <a href="{% url_for('tasks.complete', task_id =
              task.task_id) %}">Mark as Complete</a>
71           </td>
72         </tr>
73       {% endfor %}
74     </table>
75   </div>
76 </div>
77 <br>
78 <br>
79 <div class="entries">
80   <h2>Closed tasks:</h2>
81   <div class="datagrid">

```

```

82 <table>
83   <thead>
84     <tr>
85       <th width="200px"><strong>Task Name</strong></th>
86       <th width="75px"><strong>Due Date</strong></th>
87       <th width="100px"><strong>Posted Date</strong></th>
88       <th width="50px"><strong>Priority</strong></th>
89       <th width="90px"><strong>Posted By</strong></th>
90       <th><strong>Actions</strong></th>
91     </tr>
92   </thead>
93   {% for task in closed_tasks %}
94     <tr>
95       <td width="200px">{{ task.name }}</td>
96       <td width="75px">{{ task.due_date }}</td>
97       <td width="100px">{{ task.posted_date }}</td>
98       <td width="50px">{{ task.priority }}</td>
99       <td width="90px">{{ task.poster.name }}</td>
100      <td>
101        <a href="{% url_for('tasks.delete_entry', task_id =
          task.task_id) %}">Delete</a>
102      </td>
103    </tr>
104    {% endfor %}
105  </table>
106 </div>
107 <div>
108
109 {% endblock %}

```

Step 4: Updating the *main* app

With the Blueprints done, let's now turn our attention to the main project, where we will register the Blueprints.

`__init__.py`

```

1 # project/__init__.py
2

```

```

3
4 from flask import Flask
5 from flask.ext.sqlalchemy import SQLAlchemy
6
7 app = Flask(__name__)
8 app.config.from_pyfile('_config.py')
9 db = SQLAlchemy(app)
10
11 from project.users.views import users_blueprint
12 from project.tasks.views import tasks_blueprint
13
14 # register our blueprints
15 app.register_blueprint(users_blueprint)
16 app.register_blueprint(tasks_blueprint)

```

run.py

```

1 # run.py
2
3
4 from project import app
5 app.run(debug=True)

```

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 virtualenv and start over:

```

1 $ virtualenv env
2 $ source env/bin/activate
3 $ pip install -r requirements.txt

```

Now run the app:

```

1 $ 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 create a new database.

Update the Models

You just need to update the import within *models.py*.

From:

```
1 from views import db
```

To:

```
1 from project import db
```

Update *db_create.py*

```
1 # db_create.py
2
3
4 from datetime import date
5
6 from project import db
7 from project.models import Task, User
8
9 # create the database and the db table
10 db.create_all()
11
12 # insert data
13 db.session.add(
14     User("admin", "ad@min.com", "admin", "admin")
15 )
16 db.session.add(
17     Task("Finish this tutorial", date(2015, 3, 13), 10, date(2015,
18         2, 13), 1, 1)
19 )
20 db.session.add(
21     Task("Finish Real Python", date(2015, 3, 13), 10, date(2015, 2,
22         13), 1, 1)
23 )
24 # commit the changes
25 db.session.commit()
```

Run the script:

```
1 $ 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; login and logout; 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 function (individual responsibility) using Blueprints, 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*:

```
1 import os
2 import unittest
3
4 from project import app, db
5 from project._config import basedir
6 from project.models import Task, User
```

Now run the tests:

```
1 $ nosetests --with-coverage --cover-erase --cover-package=project
2 .....
3 Name                               Stmt  Miss  Cover  Missing
4 -----
5 project                             9      0  100%
6 project._config                     7      0  100%
7 project.models                      35      2   94%   30, 51
8 project.tasks                       0      0  100%
9 project.tasks.forms                 9      0  100%
10 project.tasks.views                55      0  100%
```

```
11 project.users          0      0  100%
12 project.users.forms    11      0  100%
13 project.users.views    50      0  100%
14 -----
15 TOTAL                  176      2   99%
16 -----
17 Ran 25 tests in 1.679s
18
19 OK
```

Chapter 15

Flask: FlaskTaskr, Part 6 - New features!

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
Unit 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
Boilerplate and Workflow	No
Continuous Integration and Delivery	No

Task	Complete
------	----------

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. Let's place it on the right side of the navigation bar.

Start with a test

```
1 def test_task_template_displays_logged_in_user_name(self):
2     self.register(
3         'Fletcher', 'fletcher@realpython.com', 'python101',
4         'python101'
5     )
6     self.login('Fletcher', 'python101')
7     response = self.app.get('tasks/', follow_redirects=True)
8     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.

Write Code

First, let's add the username to the session during the logging in process in `users/views.py`:

```
1 session['name'] = user.name
```

Simply add the above code to the `else` block in the `login()` function. Also, be sure to update the `logout()` function as well:

```
1 @users_blueprint.route('/logout/')
2 @login_required
3 def logout():
4     session.pop('logged_in', None)
5     session.pop('user_id', None)
6     session.pop('role', None)
7     session.pop('name', None)
8     flash('Goodbye!')
9     return redirect(url_for('users.login'))
```

Next, update the `tasks()` function within `tasks/views.py` to pass in the username to the template:

```

1 @tasks_blueprint.route('/tasks/')
2 @login_required
3 def tasks():
4     return render_template(
5         'tasks.html',
6         form=AddTaskForm(request.form),
7         open_tasks=open_tasks(),
8         closed_tasks=closed_tasks(),
9         username=session['name']
10    )

```

Finally, update the navbar within parent template in `*templates/_base.html*`:

```

1 <div class="navbar navbar-inverse navbar-fixed-top"
2     role="navigation">
3     <div class="container">
4         <div class="navbar-header">
5             <button type="button" class="navbar-toggle"
6                 data-toggle="collapse" data-target=".navbar-collapse">
7                 <span class="sr-only">Toggle navigation</span>
8                 <span class="icon-bar"></span>
9                 <span class="icon-bar"></span>
10                <span class="icon-bar"></span>
11            </button>
12            <a class="navbar-brand" href="/">FlaskTaskr</a>
13        </div>
14        <div class="collapse navbar-collapse">
15            <ul class="nav navbar-nav">
16                {% if not session.logged_in %}
17                <li><a href="{{ url_for('users.register') }}">Signup</a></li>
18                {% else %}
19                <li><a href="{{ url_for('users.logout') }}">Signout</a></li>
20                {% endif %}
21            </ul>
22            {% if session.logged_in %}
23            <ul class="nav navbar-nav navbar-right">
24                <li><a>Welcome, {{username}}.</a></li>
25            </ul>
26            {% endif %}
27        </div>
28    </div>
29 </div>

```

```

25     </div><!--/.nav-collapse -->
26 </div>
27 </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.

Let's dive right in.

Start with a test

```

1  def
2      test_users_cannot_see_task_modify_links_for_tasks_not_created_by_them(self):
3          self.register('Michael', 'michael@realpython.com',
4                        'python', 'python')
5          self.login('Michael', 'python')
6          self.app.get('tasks/', follow_redirects=True)
7          self.create_task()
8          self.logout()
9          self.register(
10              'Fletcher', 'fletcher@realpython.com', 'python101',
11              'python101'
12          )
13          response = self.login('Fletcher', 'python101')
14          self.app.get('tasks/', follow_redirects=True)
15          self.assertNotIn(b'Mark as complete', response.data)
16          self.assertNotIn(b'Delete', response.data)
17
18  def
19      test_users_can_see_task_modify_links_for_tasks_created_by_them(self):
20          self.register('Michael', 'michael@realpython.com',
21                        'python', 'python')

```

```

17     self.login('Michael', 'python')
18     self.app.get('tasks/', follow_redirects=True)
19     self.create_task()
20     self.logout()
21     self.register(
22         'Fletcher', 'fletcher@realpython.com', 'python101',
23         'python101'
24     )
25     self.login('Fletcher', 'python101')
26     self.app.get('tasks/', follow_redirects=True)
27     response = self.create_task()
28     self.assertIn(b'complete/2/', response.data)
29     self.assertIn(b'complete/2/', response.data)
30
31     def test_admin_users_can_see_task_modify_links_for_all_tasks(self):
32         self.register('Michael', 'michael@realpython.com',
33             'python', 'python')
34         self.login('Michael', 'python')
35         self.app.get('tasks/', follow_redirects=True)
36         self.create_task()
37         self.logout()
38         self.create_admin_user()
39         self.login('Superman', 'allpowerful')
40         self.app.get('tasks/', follow_redirects=True)
41         response = self.create_task()
42         self.assertIn(b'complete/1/', response.data)
43         self.assertIn(b'delete/1/', response.data)
44         self.assertIn(b'complete/2/', response.data)
45         self.assertIn(b'delete/2/', response.data)

```

Look closely at these tests before you run them. Will they both fail? Now 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.

Write Code

As for the code, we simply need to make a few changes in the *project/templates/tasks.html* template:

```

1 {% for task in open_tasks %}

```

```

2 <tr>
3   <td>{{ task.name }}</td>
4   <td>{{ task.due_date }}</td>
5   <td>{{ task.posted_date }}</td>
6   <td>{{ task.priority }}</td>
7   <td>{{ task.poster.name }}</td>
8   <td>
9     {% if task.poster.name == session.name or session.role ==
10       "admin" %}
11       <a href="{{ url_for('tasks.delete_entry', task_id =
12         task.task_id) }}">Delete</a> -
13       <a href="{{ url_for('tasks.complete', task_id =
14         task.task_id) }}">Mark as Complete</a>
15     {% else %}
16       <span>N/A</span>
17     {% endif %}
18   </td>
19 </tr>
20 {% endfor %}

```

and

```

1 {% for task in closed_tasks %}
2   <tr>
3     <td>{{ task.name }}</td>
4     <td>{{ task.due_date }}</td>
5     <td>{{ task.posted_date }}</td>
6     <td>{{ task.priority }}</td>
7     <td>{{ task.poster.name }}</td>
8     <td>
9       {% if task.poster.name == session.name or session.role ==
10         "admin" %}
11         <a href="{{ url_for('tasks.delete_entry', task_id =
12           task.task_id) }}">Delete</a>
13       {% else %}
14         <span>N/A</span>
15       {% endif %}
16     </td>
17   </tr>
18 {% 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? Using your new found knowledge of test driven development, see if you can implement it.

Next, let's take care of some housekeeping chores before we deploy the application to Heroku.

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 you should never write your own cryptographic hasher, because unless you are a cryptographer with an advanced degree in computer science or mathematics, you will probably make a mistake. Besides, 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](#):

```
1 $ pip install flask-bcrypt
2 $ pip freeze > requirements.txt
```

To set up the extension, simply import the class wrapper and pass the Flask app object into it:

```
1 # project/__init__.py
2
3
4 from flask import Flask
5 from flask.ext.sqlalchemy import SQLAlchemy
6 from flask.ext.bcrypt import Bcrypt
7
8 app = Flask(__name__)
9 app.config.from_pyfile('_config.py')
10 bcrypt = Bcrypt(app)
11 db = SQLAlchemy(app)
12
13 from project.users.views import users_blueprint
14 from project.tasks.views import tasks_blueprint
15
16 # register our blueprints
```



```

17 app.register_blueprint(users_blueprint)
18 app.register_blueprint(tasks_blueprint)

```

Then update *users/views.py* so that password is hashed during registration:

```

1 if form.validate_on_submit():
2     new_user = User(
3         form.name.data,
4         form.email.data,
5         bcrypt.generate_password_hash(form.password.data)
6     )

```

The `login()` function also needs to be updated:

```

1 @users_blueprint.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     form = LoginForm(request.form)
5     if request.method == 'POST':
6         if form.validate_on_submit():
7             user =
8                 User.query.filter_by(name=request.form['name']).first()
9             if user is not None and bcrypt.check_password_hash(
10                 user.password, request.form['password']):
11                 session['logged_in'] = True
12                 session['user_id'] = user.id
13                 session['role'] = user.role
14                 session['name'] = user.name
15                 flash('Welcome!')
16                 return redirect(url_for('tasks.tasks'))
17             else:
18                 error = 'Invalid username or password.'
19         return render_template('login.html', form=form, error=error)

```

Don't forget to update the imports:

```

1 from project import db, bcrypt

```

Manual Test Bcrypt

Delete the database. Then update the *db_create.py* script:

```

1 # db_create.py
2
3
4 from project import db
5
6 # create the database and the db table
7 db.create_all()
8
9
10 # commit the changes
11 db.session.commit()

```

Fire up the server. Register a user. Login.

Unit Test

Run the test suite:

```

1 $ nosetests --with-coverage --cover-erase --cover-package=project
2 EEE....EEE.EEE.....
3
4 Name                Stmts   Miss  Cover    Missing
5 -----
6 project              11      0   100%
7 project._config       7      0   100%
8 project.models        35      2    94%   30, 51
9 project.tasks          0      0   100%
10 project.tasks.forms    9      0   100%
11 project.tasks.views   55     19    65%   81, 93-102, 108-117
12 project.users          0      0   100%
13 project.users.forms   11      0   100%
14 project.users.views   52      0   100%
15 -----
16 TOTAL                180     21    88%
17 -----
18 Ran 29 tests in 9.979s
19
20 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. So, we need to update the following helper methods within both of the test files:

```
1 def create_user(self, name, email, password):
2     new_user = User(
3         name=name,
4         email=email,
5         password=bcrypt.generate_password_hash(password)
6     )
7     db.session.add(new_user)
8     db.session.commit()
```

and

```
1 def create_admin_user(self):
2     new_user = User(
3         name='Superman',
4         email='admin@realpython.com',
5         password=bcrypt.generate_password_hash('allpowerful'),
6         role='admin'
7     )
8     db.session.add(new_user)
9     db.session.commit()
```

Make sure to update the imports:

```
1 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 HTTPExceptions like 404 and 500. Let's look at a quick example.

First, we need to write our tests...

Start with a test

Let's add these to a new test file called `test_main.py`:

```
1 # project/test_main.py
2
3
4 import os
5 import unittest
6
7 from project import app, db
8 from project._config import basedir
9 from project.models import User
10
11
12 TEST_DB = 'test.db'
13
14
15 class MainTests(unittest.TestCase):
16
17     #####
18     ##### setup and teardown #####
19     #####
20
21     # executed prior to each test
22     def setUp(self):
23         app.config['TESTING'] = True
24         app.config['WTF_CSRF_ENABLED'] = False
25         app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + \
26             os.path.join(basedir, TEST_DB)
27         self.app = app.test_client()
28         db.create_all()
29
```

```

30     # executed after each test
31     def tearDown(self):
32         db.session.remove()
33         db.drop_all()
34
35
36     #####
37     ##### helper methods #####
38     #####
39
40     def login(self, name, password):
41         return self.app.post('/', data=dict(
42             name=name, password=password), follow_redirects=True)
43
44
45     #####
46     ##### tests #####
47     #####
48
49     def test_404_error(self):
50         response = self.app.get('/this-route-does-not-exist/')
51         self.assertEqual(response.status_code, 404)
52
53     def test_500_error(self):
54         bad_user = User(
55             name='Jeremy',
56             email='jeremy@realpython.com',
57             password='django'
58         )
59         db.session.add(bad_user)
60         db.session.commit()
61         response = self.login('Jeremy', 'django')
62         self.assertEqual(response.status_code, 500)
63
64
65 if __name__ == "__main__":
66     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.

```
1 def test_404_error(self):
2     response = self.app.get('/this-route-does-not-exist/')
3     self.assertEqual(response.status_code, 404)
4     self.assertIn(b'Sorry. There\'s nothing here.', response.data)
```

Run them 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. Let's 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:

```
1 def test_500_error(self):
2     bad_user = User(
3         name='Jeremy',
4         email='jeremy@realpython.com',
5         password='django'
6     )
7     db.session.add(bad_user)
8     db.session.commit()
9     response = self.login('Jeremy', 'django')
10    self.assertEqual(response.status_code, 500)
11    self.assertNotIn(b'ValueError: Invalid salt', response.data)
12    self.assertIn(b'Something went terribly wrong.', response.data)
```

Run the tests: FAILED (errors=1, failures=1). Nice. Let's get them passing.

404 Error Handler

View

The error handlers are set just like any other view.

Update `project/__init__.py`:

```

1 @app.errorhandler(404)
2 def not_found(error):
3     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*:

```

1 {% extends "_base.html" %}
2
3 {% block content %}
4
5     <h1>404</h1>
6     <p>Sorry. There's nothing here.</p>
7     <p><a href="{url_for('users.login')}}">Go back home</a></p>
8
9 {% 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.

500 Error Handler

View

Update *project/__init__.py*:

```

1 @app.errorhandler(500)
2 def internal_error(error):
3     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*:

```
1 {% extends "_base.html" %}
2
3 {% block content %}
4
5     <h1>500</h1>
6     <p>Something went terribly wrong. Fortunately we are working to
       fix it right now!</p>
7     <p><a href="{url_for('users.login')}}">Go back home</a></p>
8
9 {% endblock %}
```

Test

If you run the tests, you'll see we still have an error, which we'll address in a second. Let's manually test this to see what's actually happening. Temporarily update the part of the `register()` function, in *users/views.py*, where the new user is created.

Change:

```
1 new_user = User(
2     form.name.data,
3     form.email.data,
4     bcrypt.generate_password_hash(form.password.data)
5 )
```

To:

```
1 new_user = User(
2     form.name.data,
3     form.email.data,
4     form.password.data
5 )
```

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 login. 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 back up the server, register another new user, and then try to login. Now you should see our custom 500 error page.

Make sure to add `bcrypt` back to the view, `users/views.py`:

```
1 new_user = User(  
2     form.name.data,  
3     form.email.data,  
4     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

Add `DEBUG = True` to the `_config.py` file and then update `run.py`:

```
1 # run.py  
2  
3  
4 from project import app  
5 app.run()
```

Fire up the server. It should still run in debug mode. Now let's update `setUp()` for all test files:

```
1 def setUp(self):  
2     app.config['TESTING'] = True  
3     app.config['WTF_CSRF_ENABLED'] = False  
4     app.config['DEBUG'] = False  
5     app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + \  
6         os.path.join(basedir, TEST_DB)  
7     self.app = app.test_client()  
8     db.create_all()  
9  
10    self.assertEqual(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.assertEqual(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 this is later and how to set this test up right. For now, let's move on to logging all errors in our app.

Error Logging

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 potential 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*:

```
1 @app.errorhandler(404)
2 def page_not_found(error):
3     if app.debug is not True:
4         now = datetime.datetime.now()
5         r = request.url
6         with open('error.log', 'a') as f:
7             current_timestamp = now.strftime("%d-%m-%Y %H:%M:%S")
8             f.write("\n404 error at {}: {}".format(current_timestamp, r))
9     return render_template('404.html'), 404
10
11
12 @app.errorhandler(500)
13 def internal_error(error):
14     db.session.rollback()
15     if app.debug is not True:
16         now = datetime.datetime.now()
17         r = request.url
18         with open('error.log', 'a') as f:
19             current_timestamp = now.strftime("%d-%m-%Y %H:%M:%S")
```

```

20         f.write("\n500 error at {}: {}".format(current_timestamp, r))
21     return render_template('500.html'), 500

```

Be sure to update the imports as well:

```

1 import datetime
2 from flask import Flask, render_template, request
3 from flask.ext.sqlalchemy import SQLAlchemy
4 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. 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:

```

1 404 error at 29-03-2015 13:56:27:
   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:

```

1 new_user = User(
2     form.name.data,
3     form.email.data,
4     bcrypt.generate_password_hash(form.password.data)
5 )

```

To:

```

1 new_user = User(
2     form.name.data,
3     form.email.data,
4     form.password.data
5 )

```

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:

```
1 500 error at 29-03-2015 13:58:03: http://localhost:5000/
```

That's it. Make sure to add `bcrypt` back to `users/views.py` and turn debug mode back on.

Deployment Options

As far as deployment options go, [PythonAnywhere](#) and [Heroku](#) are great options. 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. Signup for [Heroku](#)
2. Login and download the [Heroku Toolbelt](#) applicable to your operating system
3. Once installed, open your command-line 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. Activate your virtualenv
5. Install [gunicorn](#): `pip install gunicorn`.
6. Heroku recognizes the dependencies needed through a *requirements.txt* file. Make sure to update you file: `pip freeze > requirements.txt`. Keep in mind that this will only create the dependencies from the libraries you installed using Pip. If you used `easy_install`, you will need to add them directly to the file.
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.
8. Update the *run.py* file. 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`:

```

1 # run.py
2
3
4 import os
5 from project import app
6
7 port = int(os.environ.get('PORT', 5000))
8 app.run(host='0.0.0.0', port=port)

```

Set debug to False within the config file. Why? 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:

```
1 $ heroku create
```

Deploy your code to Heroku:

```
1 $ git push heroku master
```

Add a PostgreSQL database:

```
1 $ heroku addons:create heroku-postgresql:dev
```

Check to make sure your app is running:

```
1 $ heroku ps
```

View the app in your browser:

```
1 $ heroku open
```

If you see errors, open the Heroku log to view all errors and output:

```
1 $ heroku logs
```

Finally, run your tests on Heroku:

```
1 $ heroku run nosetests
2 Running `nosetests` attached to terminal... up, run.9405
3 .....
4 -----
5 Ran 30 tests in 24.469s
6
7 OK
```

All should pass.

That's it. Make sure that you also PUSH your local repository to Github.

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 (March 29, 2015), Fabric only works with Python 2.x. It's in the process of being ported to Python 3, which should be finished in mid-2015.

Setup

As always, start by installing the dependency with Pip:

```
1 $ pip install fabric==1.10.1
2 $ 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, then place it in 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*:

```

1 def test():
2     local("nosetests -v")
3
4
5 def commit():
6     message = raw_input("Enter a git commit message: ")
7     local("git add . && git commit -am '{}'.format(message)")
8
9
10 def push():
11     local("git push origin master")
12
13
14 def prepare():
15     test()
16     commit()
17     push()

```

Essentially, we import the `local` method from Fabric, then run the basic shell commands for testing and PUSHing to GitHub as you've seen before.

Test this out:

```

1 $ 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:

```

1 def test():
2     with settings(warn_only=True):
3         result = local("nosetests -v", capture=True)
4         if result.failed and not confirm("Tests failed. Continue?"):
5             abort("Aborted at user request.")

```

Also update the imports:

```
1 from fabric.api import local, settings, abort
2 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:

```
1 def pull():
2     local("git pull origin master")
3
4 def heroku():
5     local("git push heroku master")
6
7 def heroku_test():
8     local("heroku run nosetests -v")
9
10 def deploy():
11     pull()
12     test()
13     commit()
14     heroku()
15     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.

NOTE: This is a relatively basic fabfile. If are working with a team of developers and PUSH code to multiple servers, then your fabfile will be much larger and more complex.

The last thing we need to add is a quick rollback.

```
1 def rollback():
2     local("heroku rollback")
```

Updated file:

```

1 from fabric.api import local, settings, abort
2 from fabric.contrib.console import confirm
3
4
5 # prep
6
7 def test():
8     with settings(warn_only=True):
9         result = local("nosetests -v", capture=True)
10        if result.failed and not confirm("Tests failed. Continue?"):
11            abort("Aborted at user request.")
12
13
14 def commit():
15     message = raw_input("Enter a git commit message: ")
16     local("git add . && git commit -am '{}'.format(message))
17
18
19 def push():
20     local("git push origin master")
21
22
23 def prepare():
24     test()
25     commit()
26     push()
27
28
29 # deploy
30
31 def pull():
32     local("git pull origin master")
33
34
35 def heroku():
36     local("git push heroku master")
37
38
39 def heroku_test():
40     local("heroku run nosetests -v")

```

```
41
42
43 def deploy():
44     # pull()
45     test()
46     # commit()
47     heroku()
48     heroku_test()
49
50
51 # rollback
52
53 def rollback():
54     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:

```
1 $ 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.

Building a REST API

Finally, let's take a quick look at how to design a RESTful API in Flask using FlaskTaskr. 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. 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 REST API, you expose much of your internal system to the rest of the world (or even other areas within your own organization). Keep this in mind. Only allows users or programs access to a limited set of data, using only the needed HTTP methods. Never expose more than necessary.

Basic REST Design Practices

URLs (endpoints) are used for identifying a specific resource, while the HTTP methods define the actions one can take on those resources. Each resource should only have 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 new task	Update all tasks	Delete all tasks
/tasks/<id>/	View specific task	N/A	Update specific task	Delete specific 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 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 front-end 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 Backbone, 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*. Now add the following code:

```
1 # tests/test_api.py
2
3
4 import os
5 import unittest
6 from datetime import date
7
8 from project import app, db
9 from project._config import basedir
10 from project.models import Task
11
12
13 TEST_DB = 'test.db'
14
15
16 class APITests(unittest.TestCase):
17
18     #####
19     ##### setup and teardown #####
20     #####
21
22     # executed prior to each test
23     def setUp(self):
24         app.config['TESTING'] = True
25         app.config['WTF_CSRF_ENABLED'] = False
26         app.config['DEBUG'] = False
27         app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + \
28             os.path.join(basedir, TEST_DB)
29         self.app = app.test_client()
30         db.create_all()
31
32         self.assertEqual(app.debug, False)
33
34     # executed after each test
35     def tearDown(self):
```



```

36         db.session.remove()
37         db.drop_all()
38
39         #####
40         ##### helper methods #####
41         #####
42
43     def add_tasks(self):
44         db.session.add(
45             Task(
46                 "Run around in circles",
47                 date(2015, 10, 22),
48                 10,
49                 date(2015, 10, 5),
50                 1,
51                 1
52             )
53         )
54         db.session.commit()
55
56         db.session.add(
57             Task(
58                 "Purchase Real Python",
59                 date(2016, 2, 23),
60                 10,
61                 date(2016, 2, 7),
62                 1,
63                 1
64             )
65         )
66         db.session.commit()
67
68         #####
69         ##### tests #####
70         #####
71
72     def test_collection_endpoint_returns_correct_data(self):
73         self.add_tasks()
74         response = self.app.get('api/v1/tasks/',
75                                 follow_redirects=True)

```

```

75         self.assertEqual(response.status_code, 200)
76         self.assertEqual(response.mimetype, 'application/json')
77         self.assertIn(b'Run around in circles', response.data)
78         self.assertIn(b'Purchase Real Python', response.data)
79
80
81 if __name__ == "__main__":
82     unittest.main()

```

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.

Write the Code

Start by creating a new Blueprint called “api”. You just need two files - `__init__.py` and `views.py`. Add the following code to `views.py`:

```

1 # project/api/views.py
2
3
4 from functools import wraps
5 from flask import flash, redirect, jsonify, \
6     session, url_for, Blueprint
7
8 from project import db
9 from project.models import Task
10
11
12 #####
13 #### config ####
14 #####
15

```

```

16 api_blueprint = Blueprint('api', __name__)
17
18
19 #####
20 ##### helper functions #####
21 #####
22
23 def login_required(test):
24     @wraps(test)
25     def wrap(*args, **kwargs):
26         if 'logged_in' in session:
27             return test(*args, **kwargs)
28         else:
29             flash('You need to login first.')
30             return redirect(url_for('users.login'))
31     return wrap
32
33
34 def open_tasks():
35     return db.session.query(Task).filter_by(
36         status='1').order_by(Task.due_date.asc())
37
38
39 def closed_tasks():
40     return db.session.query(Task).filter_by(
41         status='0').order_by(Task.due_date.asc())
42
43
44 #####
45 ##### routes #####
46 #####
47
48 @api_blueprint.route('/api/v1/tasks/')
49 def api_tasks():
50     results = db.session.query(Task).limit(10).offset(0).all()
51     json_results = []
52     for result in results:
53         data = {
54             'task_id': result.task_id,
55             'task name': result.name,

```

```

56         'due date': str(result.due_date),
57         'priority': result.priority,
58         'posted date': str(result.posted_date),
59         'status': result.status,
60         'user id': result.user_id
61     }
62     json_results.append(data)
63     return jsonify(items=json_results)

```

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. Take a minute to read about the 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:

```

1 return json.dumps(data), 200, { "Content-Type" :
    "application/json"}

```

Make sure to register the Blueprint in `project/__init__.py`:

```

1 from project.users.views import users_blueprint
2 from project.tasks.views import tasks_blueprint
3 from project.api.views import api_blueprint
4
5
6 # register our blueprints
7 app.register_blueprint(users_blueprint)
8 app.register_blueprint(tasks_blueprint)
9 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?

```
1 $ nosetests tests/test_api.py
2 .
3 -----
4 Ran 1 test in 0.050s
5
6 OK
```

Of course!

NOTE: Notice 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

```
1 def test_resource_endpoint_returns_correct_data(self):
2     self.add_tasks()
3     response = self.app.get('api/v1/tasks/2', follow_redirects=True)
4     self.assertEqual(response.status_code, 200)
5     self.assertEqual(response.mimetype, 'application/json')
6     self.assertIn(b'Purchase Real Python', response.data)
7     self.assertNotIn(b'Run around in circles', response.data)
```

This is very similar to our first test.

Write the Code

```
1 @api_blueprint.route('/api/v1/tasks/<int:task_id>')
2 def task(task_id):
3     result =
4         db.session.query(Task).filter_by(task_id=task_id).first()
5     json_result = {
```

```

5         'task_id': result.task_id,
6         'task_name': result.name,
7         'due date': str(result.due_date),
8         'priority': result.priority,
9         'posted date': str(result.posted_date),
10        'status': result.status,
11        'user id': result.user_id
12    }
13    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

```

1 $ nosetests tests/test_api.py
2 ..
3 -----
4 Ran 2 tests in 0.068s
5
6 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 more user-friendly error. Let's update the code.

Updated Endpoints

Test

```

1 def test_invalid_resource_endpoint_returns_error(self):
2     self.add_tasks()
3     response = self.app.get('api/v1/tasks/209',
4                             follow_redirects=True)
5     self.assertEqual(response.status_code, 404)
6     self.assertEqual(response.mimetype, 'application/json')
7     self.assertIn(b'Element does not exist', response.data)

```

Write the Code

```
1 @api_blueprint.route('/api/v1/tasks/<int:task_id>')
2 def task(task_id):
3     result =
4         db.session.query(Task).filter_by(task_id=task_id).first()
5     if result:
6         json_result = {
7             'task_id': result.task_id,
8             'task name': result.name,
9             'due date': str(result.due_date),
10            'priority': result.priority,
11            'posted date': str(result.posted_date),
12            'status': result.status,
13            'user id': result.user_id
14        }
15        return jsonify(items=json_result)
16    else:
17        result = {"error": "Element does not exist"}
18        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:

```
1 @api_blueprint.route('/api/v1/tasks/<int:task_id>')
2 def task(task_id):
3     result =
4         db.session.query(Task).filter_by(task_id=task_id).first()
5     if result:
6         result = {
7             'task_id': result.task_id,
8             'task name': result.name,
9             'due date': str(result.due_date),
10            'priority': result.priority,
11            'posted date': str(result.posted_date),
12            'status': result.status,
```

```

12         'user id': result.user_id
13     }
14     code = 200
15 else:
16     result = {"error": "Element does not exist"}
17     code = 404
18     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.

```

1 $ nosetests tests/test_api.py
2 ...
3 -----
4 Ran 3 tests in 0.100s
5
6 OK

```

Let's also test the entire test suite to ensure we didn't break anything anywhere else in the codebase:

```

1 $ nosetests --with-coverage --cover-erase --cover-package=project
2 .....
3

```

Name	Stmts	Miss	Cover	Missing
project	31	8	74%	40-47
project._config	8	0	100%	
project.api	0	0	100%	
project.api.views	31	8	74%	24-31, 35, 40
project.models	35	2	94%	30, 51
project.tasks	0	0	100%	
project.tasks.forms	9	0	100%	
project.tasks.views	55	0	100%	
project.users	0	0	100%	
project.users.forms	11	0	100%	


```
15 project.users.views      52      0    100%
16 -----
17 TOTAL                    232     18    92%
18 -----
19 Ran 33 tests in 20.022s
20
21 OK
```

Perfect.

Deploy

Turn off debug mode, then run the following commands:

```
1 $ fab prepare
2 $ fab deploy
```

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 setup 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.

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

1. Navigate to a directory out of your “realpython” directory. The “desktop” or “documents” directories are good choices since they are easy to access.
2. [Clone](#) the boilerplate template from Github:

```
1 $ git clone https://github.com/mjhea0/flask-boilerplate.git
2 $ cd flask-boilerplate
```

This creates a new folder called “flask-boilerplate”.

Project structure:

```
1
2 Procfile
3 Procfile.dev
4 README.md
5 app.py
6 config.py
7 error.log
8 forms.py
9 models.py
10 requirements.txt
11 static
12     CSS
13         bootstrap-3.0.0.min.css
14         bootstrap-theme-3.0.0.css
15         bootstrap-theme-3.0.0.min.css
16         font-awesome-3.2.1.min.css
17         layout.forms.css
18         layout.main.css
19         main.css
```

```

20     main.quickfix.css
21     main.responsive.css
22 font
23     FontAwesome.otf
24     fontawesome-webfont.eot
25     fontawesome-webfont.svg
26     fontawesome-webfont.ttf
27     fontawesome-webfont.woff
28 ico
29     apple-touch-icon-114-precomposed.png
30     apple-touch-icon-144-precomposed.png
31     apple-touch-icon-57-precomposed.png
32     apple-touch-icon-72-precomposed.png
33     favicon.png
34 img
35 js
36     libs
37         bootstrap-3.0.0.min.js
38         jquery-1.10.2.min.js
39         modernizr-2.6.2.min.js
40         respond-1.3.0.min.js
41     plugins.js
42     script.js
43 templates
44     errors
45         404.html
46         500.html
47     forms
48         forgot.html
49         login.html
50         register.html
51     layouts
52         form.html
53         main.html
54     pages
55         placeholder.about.html
56         placeholder.home.html

```

3. Install the various libraries and dependencies:

```
1 $ pip install -r requirements.txt
```

You can also view the dependencies by running the command `pip freeze`:

```
1 Fabric==1.8.2
2 Flask==0.10.1
3 Flask-SQLAlchemy==1.0
4 Flask-WTF==0.9.4
5 Jinja2==2.7.2
6 MarkupSafe==0.18
7 SQLAlchemy==0.9.3
8 WTForms==1.0.5
9 Werkzeug==0.9.4
10 coverage==3.7.1
11 ecdsa==0.10
12 itsdangerous==0.23
13 paramiko==1.12.2
14 pycrypto==2.6.1
15 wsgiref==0.1.2
```

Deploy to Heroku

Before you do this delete the hidden “.git” directory. Follow the steps in the deployment lesson. Or view the Github readme [here](#)

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: Remember: the fabfile performs this exact workflow (steps 1 through 5) for you automatically

That's it. You now have a skeleton app to work with to build your own applications. Cheers!

NOTE: If you prefer PythonAnywhere over Heroku, simply deploy your app there during the setup phase and then change step #4 in your workflow to deploy to PythonAnywhere instead of Heroku. Simple. You can also look at the documentation [here](#).

Chapter 16

Flask: FlaskTaskr, Part 7: 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 (regressions). In practice, members of a development team integrate (via Github) their code frequently, which is 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 on Github where 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 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 over them. However, they take longer to setup, since you have to host them yourself, and you have to continue to maintain them. The hosted services, on the other hand, are super easy to setup and 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 setup on Github and Heroku, a local Git repository, and a repository setup 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”. At that point, 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](#).

Once logged in, navigate to your profile page. You should see all the open 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. 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.

```
1 # specify language
2 language:
3   - python
4
5 # specify python version(s)
6 python:
7   - "3.4"
8   - "2.7"
9
10 # install dependencies
11 install:
```

```
12 - pip install -r requirements.txt
13
14 # run tests
15 script:
16 - nosetests
```

Add this to a file called `.travis.yml` to the main directory. So we specified the language, the versions of Python we want to use for the tests, 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, then push to Github. This will add a new build into the Travis CI queue. Now we must wait for a worker to open up before the tests run. This can take a few minutes. Be patient.

After the tests run - and pass - it's common to place the build 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 the [Ruby Version Manager](#):

```
1 $ curl -sSL https://get.rvm.io | bash -s stable --ruby
2 $ rvm install 1.9.3
3 $ rvm use 1.9.3
4 $ rvm rubygems latest
```

Once you have Ruby installed, run `gem install travis` to install the Travis Command Line Client.

Now, within your project directory, run the following command:

```
1 $ travis setup heroku
```

This command automatically adds the necessary configuration for deploying to Heroku to the deploy section on the *.travis.yml* file, which should look something like this:

```
1 deploy:
2   provider: heroku
3   api_key:
4     secure: Dxddd3y/i7oTTsC5IHUebG/xVJlIrGbZ1CCHm9KwE56
5   app: flasktaskr_project
6   on:
7     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 recap 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, we trigger a deploy to Heroku.

With everything setup, 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, deploys 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 a 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:

```
1 # specify language
2 language:
3   - python
4
5 # specify python version(s)
6 python:
7   - "3.4"
8   - "2.7"
9
10 # install dependencies
11 install:
12   - pip install -r requirements.txt
13
14 # run tests
15 script:
16   - nosetests
17
18 # deploy!
19 deploy:
20   provider: heroku
21   api_key:
22     secure: Kq95wr8v6rGlspuTt3Y8NdPse2eandSAF0DGmQm
23   app: app_name
24   on:
25     branch: master
26     python: '3.4'
```

```
27 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:

```
1 $ 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*:

```
1 def test_index(self):
2     """ Ensure flask was set up correctly. """
3     response = self.app.get('/', content_type='html/text')
4     self.assertEqual(response.status_code, 200)
```

Run the tests. Commit the changes locally. Then push to Github.

```
1 $ git add -A
2 $ git commit -am "added tests"
3 $ 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:

```
1 @users_blueprint.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     form = LoginForm(request.form)
5     if request.method == 'POST':
6         if form.validate_on_submit():
7             user =
8                 User.query.filter_by(name=request.form['name']).first()
9             if user is not None and bcrypt.check_password_hash(
10                 user.password, request.form['password']):
11                 session['logged_in'] = True
12                 session['user_id'] = user.id
13                 session['role'] = user.role
14                 session['name'] = user.name
15                 flash('Welcome!')
16                 return redirect(url_for('tasks.tasks'))
17             else:
18                 error = 'Invalid username or password.'
```

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:

```
1 def push():
2     local("git branch")
3     branch = raw_input("Which branch do you want to push to? ")
4     local("git push origin {}".format(branch))
```

Now we can test the functionally:

```
1 $ fab prepare
2 fab prepare
3 [localhost] local: nosetests -v
4 Enter a git commit message: fabric test
5 [localhost] local: git add -A && git commit -am 'fabric test'
6 [unit-tests 5142eff] fabric test
7 3 files changed, 8 insertions(+), 29 deletions(-)
8 [localhost] local: git branch
9     master
10 * unit-tests
11 Which branch do you want to push to? unit-tests
12 [localhost] local: git push origin unit-tests
13 Counting objects: 9, done.
14 Delta compression using up to 4 threads.
15 Compressing objects: 100% (5/5), done.
16 Writing objects: 100% (5/5), 611 bytes, done.
17 Total 5 (delta 4), reused 0 (delta 0)
18 To git@github.com:realpython/flasktaskr_project.git
19     c1d508c..5142eff  unit-tests -> unit-tests
20
21 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> 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
Unit 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 REST API, and
- Add integration tests.

Patience. Keep practicing.

With that, let's move on to Behavior Driven Development!

Chapter 17

Flask: Behavior-Driven Development with Behave

The goal of this next tutorial is to introduce you to a powerful technique for developing and testing your web applications called behavior-driven development (abbreviated BDD). The application we're going to build is simple yet complex enough for you to learn how powerful BDD can be. You've probably seen this app before.

As many of you 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. More specifically, it's 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 (abbreviated 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:

```
1 Scenario: successful login
2   Given flaskr is set up
3     When we log in with "admin" and "admin"
4     Then we should see the alert "You were logged in"
```

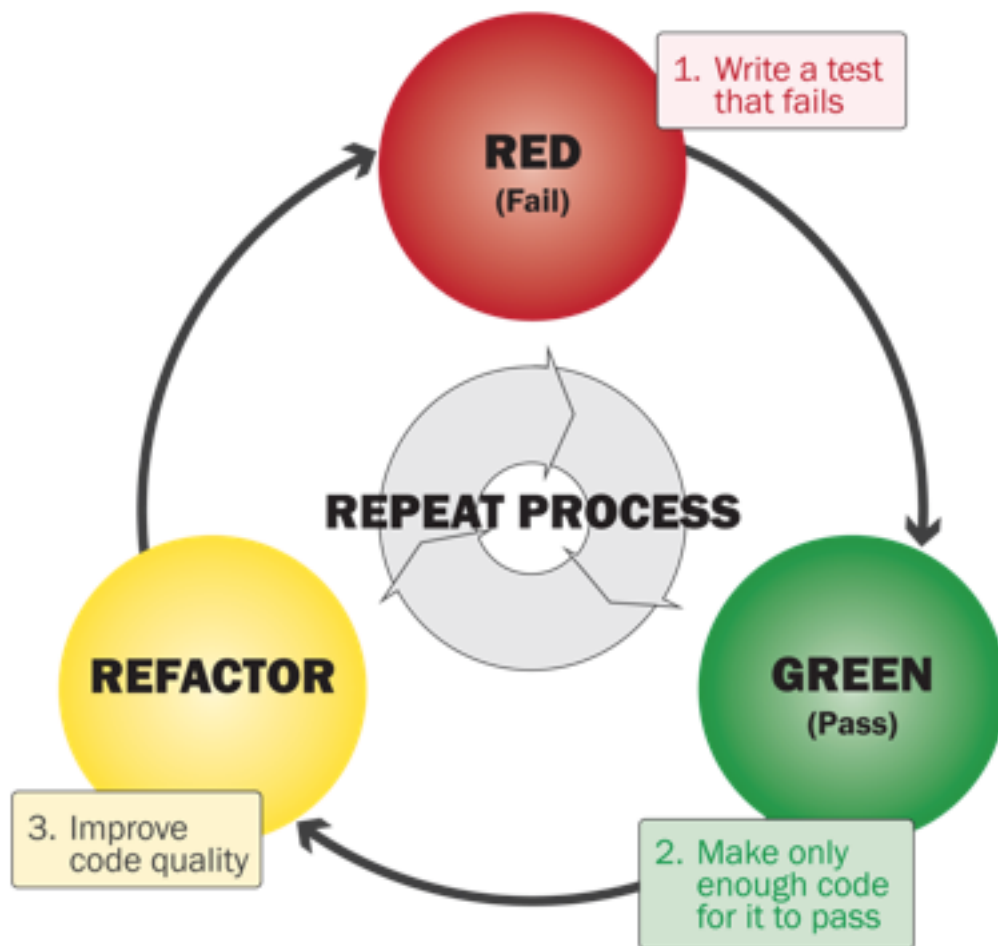


Figure 17.1: Test-Driven Development Cycle

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 login, 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: QuickStart* 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 virtualenv.
3. Set up the following files and directories:

```
1
2  app.py
3  static
4      css
5      img
6      js
7  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 “exercises” directory in the Real Python [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`

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 even 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 tremendously 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 notably - 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.

1. Create a new directory called “features”, which will eventually house all of our features files as well the subsequent tests, called steps.
2. 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](#):

```
1 Feature: flaskr is secure in that users must log in and log out to
   access certain features
2
3 Scenario: successful login
4   Given flaskr is set up
5     When we log in with "admin" and "admin"
6     Then we should see the alert "You were logged in"
7
8 Scenario: incorrect username
9   Given flaskr is set up
10    When we log in with "notright" and "admin"
11    Then we should see the alert "Invalid username"
12
13 Scenario: incorrect password
14   Given flaskr is set up
15     When we log in with "admin" and "notright"
16     Then we should see the alert "Invalid password"
17
18 Scenario: successful logout
19   Given flaskr is set up
20   and we log in with "admin" and "admin"
21   When we logout
22   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.

3. Create a “steps” directory within the “features” directory, then add a file called *auth_steps.py*.

4. Within *auth_steps.py*, let's add our first test:

```
1 from behave import *
2
3 @given(u'flaskr is set up')
4 def flask_is_setup(context):
5     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](#).

5. Run Behave:

```
1 $ behave
```

Search through the output until you find the overall stats on what features and scenarios fail:

```
1 Failing scenarios:
2   features/auth.feature:3  successful login
3   features/auth.feature:8  incorrect username
4   features/auth.feature:13 incorrect password
5   features/auth.feature:18 successful logout
6
7 0 features passed, 1 failed, 0 skipped
8 0 scenarios passed, 4 failed, 0 skipped
9 0 steps passed, 4 failed, 0 skipped, 9 undefined
10 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.

1. Create an *environment.py* file within your “features” folder
2. Add the following code:

```
1 import os
2 import sys
3
4 # add module to syspath
5 pwd = os.path.abspath(os.path.dirname(__file__))
6 project = os.path.basename(pwd)
7 new_path = pwd.strip(project)
8 full_path = os.path.join(new_path, 'flaskr')
9
10 try:
11     from flaskr import app
12 except ImportError:
13     sys.path.append(full_path)
14     from flaskr import app
15
16
17 def before_feature(context, feature):
18     context.client = app.test_client()
```

What's going on?

1. First, take a look at the function. [Flask](#) provides a handy method, `test_client()`, to use during testing to essentially mock the actual application.
2. 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*:

```
1 app` variable - `app = Flask(__name__)
```

We need access to that variable.

1. We can't just import that app since 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`:

```
1 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.

```
1 Failing scenarios:
2   features/auth.feature:3  successful login
3   features/auth.feature:8  incorrect username
4   features/auth.feature:13 incorrect password
5   features/auth.feature:18 successful logout
6
7 0 features passed, 1 failed, 0 skipped
8 0 scenarios passed, 4 failed, 0 skipped
9 4 steps passed, 0 failed, 0 skipped, 9 undefined
10 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:

```
1 @given(u'flaskr is set up')
2 def flask_is_set_up(context):
3     assert context.client
4
5 @given(u'we log in with "{username}" and "{password}"')
6 @when(u'we log in with "{username}" and "{password}"')
7 def login(context, username, password):
8     context.page = context.client.post(
9         '/login',
10        data=dict(username=username, password=password),
11        follow_redirects=True
12    )
13    assert context.page
14
15 @when(u'we log out')
16 def logout(context):
17     context.page = context.client.get('/logout',
18        follow_redirects=True)
19    assert context.page
20
21 @then(u'we should see the alert "{message}"')
22 def message(context, message):
23     assert 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, stop following 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 login and logout 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

1. Update *flaskr.py*:

```
1 @app.route('/login', methods=['GET', 'POST'])
2 def login():
3     """User login/authentication/session management."""
4     error = None
5     if request.method == 'POST':
6         if request.form['username'] != app.config['USERNAME']:
7             error = 'Invalid username'
8         elif request.form['password'] != app.config['PASSWORD']:
9             error = 'Invalid password'
10        else:
11            session['logged_in'] = True
12            flash('You were logged in')
13            return redirect(url_for('index'))
14    return render_template('login.html', error=error)
15
16
17 @app.route('/logout')
18 def logout():
19     """User logout/authentication/session management."""
20     session.pop('logged_in', None)
21     flash('You were logged out')
22     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](#)?

2. Add the following imports and update the config:

```
1 # imports
2 from flask import Flask, render_template, request, session
3
4 # configuration
5 DATABASE = ''
```

```

6 USERNAME = 'admin'
7 PASSWORD = 'admin'
8 SECRET_KEY = 'change me'

```

3. Add the *login.html* template:

```

1 <div class="container">
2   <h1>Flask - BDD</h1>
3   {% for message in get_flashed_messages() %}
4     <div class="flash">{{ message }}</div>
5   {% endfor %}
6   <h3>Login</h3>
7   {% if error %}<p class="error"><strong>Error:</strong> {{ error
8     }}{% endif %}
9   <form action="{{ url_for('login') }}" method="post">
10     <dl>
11       <dt>Username:
12       <dd><input type="text" name="username">
13       <dt>Password:
14       <dd><input type="password" name="password">
15       <br>
16       <dd><input type="submit" class="btn btn-default"
17         value="Login">
18       <span>Use "admin" for the username and password</span>
19     </dl>
20   </form>
21 </div>

```

4. While we're at it, let's update the styles; update the head of *base.html*:

```

1 <head>
2   <title>Flask - Behavior Driven Development</title>
3   <meta name="viewport" content="width=device-width,
4     initial-scale=1.0">
5   <link href="http://netdna.bootstrapcdn.com/
6     bootstrap/3.1.1/css/bootstrap.min.css" rel="stylesheet"
7     media="screen">
8   <script
9     src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
10   <script src="http://netdna.bootstrapcdn.com/

```

```

8     bootstrap/3.1.1/js/bootstrap.min.js"></script>
9 </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.

5. Finally, update *index.html*:

```

1 {% extends "base.html" %}
2
3 {% block content %}
4
5 <div class="container">
6   <h1>Flask - BDD</h1>
7   {% if not session.logged_in %}
8     <a href="{{ url_for('login') }}">log in</a>
9   {% else %}
10    <a href="{{ url_for('logout') }}">log out</a>
11  {% endif %}
12  {% for message in get_flashed_messages() %}
13    <div class="flash">{{ message }}</div>
14  {% endfor %}
15  {% if session.logged_in %}
16    <h1>Hi!</h1>
17  {% endif %}
18 </div>
19
20 {% endblock %}

```

6. Now, before we run Behave, let's do some manual testing. Run the server. Test logging in and logging out. 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 1 feature passed, 0 failed, 0 skipped
2 4 scenarios passed, 0 failed, 0 skipped
3 13 steps passed, 0 failed, 0 skipped, 0 undefined
4 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*:

```
1 Scenario: successful post
2   Given flaskr is setup
3   and we log in with "admin" and "admin"
4     When we add a new entry with "test" and "test" as the title and
      text
5     Then we should see the alert "New entry was successfully posted"
6
7 Scenario: unsuccessful post
8   Given flaskr is setup
9   Given we are not logged in
10    When we add a new entry with "test" and "test" as the title and
      text
11    Then we should see a "405" status code
```

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 an awful [code smell](#).

First Step

Add the following step to *auth.steps*:

```
1 @when(u'we add a new entry with "{title}" and "{text}" as the title
   and text')
2 def add(context, title, text):
3     context.page = context.client.post(
4         '/add',
5         data=dict(title=title, text=text),
6         follow_redirects=True
7     )
8     assert context.page
```

Run Behave

```
1 Failing scenarios:
2   features/auth.feature:24  successful post
3   features/auth.feature:30  unsuccessful post
4
5 0 features passed, 1 failed, 0 skipped
6 4 scenarios passed, 2 failed, 0 skipped
7 17 steps passed, 1 failed, 1 skipped, 2 undefined
8 Took 0m0.050s
```

Let's get this scenario, successful post, to pass.

Update *flaskr.py*

```
1 @app.route('/add', methods=['POST'])
2 def add_entry():
3     flash('New entry was successfully posted')
```

```
4     return redirect(url_for('index'))
```

Update *index.html*

```
1 {% extends "base.html" %}
2
3 {% block content %}
4
5 <div class="container">
6     <h1>Flask - BDD</h1>
7     {% if not session.logged_in %}
8         <a href="{{ url_for('login') }}">log in</a>
9     {% else %}
10        <a href="{{ url_for('logout') }}">log out</a>
11    {% endif %}
12    {% for message in get_flashed_messages() %}
13        <div class="flash">{{ message }}</div>
14    {% endfor %}
15    {% if session.logged_in %}
16        <br><br>
17        <form action="{{ url_for('add_entry') }}" method="post"
18            class="add-entry">
19            <dl>
20                <dt>Title:
21                <dd><input type="text" size="30" name="title">
22                <dt>Text:
23                <dd><textarea name="text" rows="5" cols="40"></textarea>
24                <br>
25                <dd><input type="submit" class="btn btn-default"
26                    value="Share">
27            </dd>
28        </dl>
29    </form>
30    {% endif %}
31 </div>
32 {% endblock %}
```

Run Behave

```

1 Failing scenarios:
2   features/auth.feature:30  unsuccessful post
3
4 0 features passed, 1 failed, 0 skipped
5 5 scenarios passed, 1 failed, 0 skipped
6 18 steps passed, 0 failed, 1 skipped, 2 undefined
7 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:

```

1 @app.route('/add', methods=['POST'])
2 def add_entry():
3     if not session.get('logged_in'):
4         abort(401)
5     flash('New entry was successfully posted')
6     return redirect(url_for('index'))

```

Be sure to update the imports as well:

```

1 # imports
2 from flask import (Flask, render_template, request,
3     session, flash, redirect, url_for, abort)

```

Run Behave

Looks good!

```

1 1 feature passed, 0 failed, 0 skipped
2 6 scenarios passed, 0 failed, 0 skipped
3 21 steps passed, 0 failed, 0 skipped, 0 undefined
4 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:

```
1 Scenario: successful post
2   Given flaskr is set up
3   and we log in with "admin" and "admin"
4   When we add a new entry with "test" and "test" as the title and
      text
5   Then we should see the alert "New entry was successfully posted"
6   Then we should see the post with "test" and "test" as the title
      and text
```

First Step

```
1 @then('we should see the post with "{title}" and "{text}" as the
   title and text')
2 def entry(context, title, text):
3     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

```
1 Failing scenarios:
2   features/auth.feature:24  successful post
3
4 0 features passed, 1 failed, 0 skipped
5 5 scenarios passed, 1 failed, 0 skipped
6 21 steps passed, 1 failed, 0 skipped, 0 undefined
7 Took 0m0.058s
```

This failed as expected. Now let's get the step to pass.

Create a new file called *schema.sql*:

```
1 drop table if exists entries;
2 create table entries (
3   id integer primary key autoincrement,
4   title text not null,
5   text text not null
6 );
```

Here we are setting up a single table with three fields - "id", "title", and "text".

Update *flaskr.py*

1. Add two new imports, `sqlite3` and `g`:

```
1 import sqlite3
2 from flask import (Flask, render_template, request,
3   session, flash, redirect, url_for, abort, g)
```

2. Update the config section:

```
1 # configuration
2 DATABASE = 'flaskr.db'
3 USERNAME = 'admin'
4 PASSWORD = 'admin'
5 SECRET_KEY = 'change me'
```

3. Add the following functions for managing the database connections:

```

1  # connect to database
2  def connect_db():
3      rv = sqlite3.connect(app.config['DATABASE_PATH'])
4      rv.row_factory = sqlite3.Row
5      return rv
6
7  # create the database
8  def init_db():
9      with app.app_context():
10         db = get_db()
11         with app.open_resource('schema.sql', mode='r') as f:
12             db.cursor().executescript(f.read())
13         db.commit()
14
15  # open database connection
16  def get_db():
17      if not hasattr(g, 'sqlite_db'):
18         g.sqlite_db = connect_db()
19      return g.sqlite_db
20
21  # close database connection
22  @app.teardown_appcontext
23  def close_db(error):
24      if hasattr(g, 'sqlite_db'):
25         g.sqlite_db.close()

```

4. Finally, update the `index()` and `add_entry()` functions:

```

1  @app.route('/')
2  def index():
3      db = get_db()
4      cur = db.execute('select title, text from entries order by id
5                        desc')
6      entries = cur.fetchall()
7      return render_template('index.html', entries=entries)
8
9  @app.route('/add', methods=['POST'])
10 def add_entry():
11     if not session.get('logged_in'):
12         abort(405)

```

```

12     db = get_db()
13     db.execute('insert into entries (title, text) values (?, ?)',
14                 [request.form['title'], request.form['text']])
15     db.commit()
16     flash('New entry was successfully posted')
17     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

```

1 $ python
2 >>> from flaskr import init_db
3 >>> init_db()
4 >>>

```

Update *index.html*

```

1 {% extends "base.html" %}
2
3 {% block content %}
4
5 <div class="container">
6   <h1>Flask - BDD</h1>
7   {% if not session.logged_in %}
8     <a href="{{ url_for('login') }}">log in</a>
9   {% else %}
10    <a href="{{ url_for('logout') }}">log out</a>
11  {% endif %}
12  {% for message in get_flashed_messages() %}
13    <div class="flash">{{ message }}</div>
14  {% endfor %}
15  {% if session.logged_in %}
16    <br><br>
17    <form action="{{ url_for('add_entry') }}" method="post"
      class="add-entry">

```

```

18     <dl>
19         <dt>Title:
20         <dd><input type="text" size="30" name="title">
21         <dt>Text:
22         <dd><textarea name="text" rows="5" cols="40"></textarea>
23         <br>
24         <br>
25         <dd><input type="submit" class="btn btn-default"
26             value="Share">
27     </dl>
28 </form>
29 {% endif %}
30 <ul>
31     {% for entry in entries %}
32     <li><h2>{{ entry.title }}</h2>{{ entry.text|safe }}
33     {% else %}
34     <li><em>No entries yet. Add some!</em>
35     {% endfor %}
36 </ul>
37 </div>
38 {% endblock %}

```

This adds a loop to loop through each entry, posting the title and text of each.

Environment Control

Update...

```

1  import os
2  import sys
3  import tempfile
4
5  # add module to syspath
6  pwd = os.path.abspath(os.path.dirname(__file__))
7  project = os.path.basename(pwd)
8  new_path = pwd.strip(project)
9  full_path = os.path.join(new_path, 'flaskr')
10
11  try:
12      from flaskr import app, init_db

```

```

13 except ImportError:
14     sys.path.append(full_path)
15     from flaskr import app, init_db
16
17
18 def before_feature(context, feature):
19     app.config['TESTING'] = True
20     context.db, app.config['DATABASE'] = tempfile.mkstemp()
21     context.client = app.test_client()
22     init_db()
23
24
25 def after_feature(context, feature):
26     os.close(context.db)
27     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 1 feature passed, 0 failed, 0 skipped
2 6 scenarios passed, 0 failed, 0 skipped
3 22 steps passed, 0 failed, 0 skipped, 0 undefined
4 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:

```

1 @given(u'flaskr is setup')

```

```
2 def flask_is_set_up(context):  
3     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. Ask for help on the message board, if needed. Or Google. Or look at BDD examples on GitHub. 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.

Chapter 18

Interlude: Web Frameworks, Compared

Overview

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), and
- 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: 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 classified as either a full (high-level), or micro (low-level), depending on the amount and level of automation it can perform, and its number of pre-installed components (batteries). Full frameworks come with many pre-installed batteries and a lot of low-level task automation, while micro frameworks come with few batteries and less automation. All web frameworks do offer some automation, however, 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.

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!

Chapter 19

web2py: Quick Start

Overview

[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.



Figure 19.1: web2py Logo

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 all 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. Google-it-first.

Installation

Quick Install

NOTE: This course utilizes web2py version [2.8.2](#).

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 developing from the command line, so follow the full installation guide below.

Full Install

1. Create a directory called “web2py”. Download the source code from the web2py [repository](#) and place it into this new directory. Unzip the file. Rename the new directory from the zip file to “start”. Navigate into that directory. *Both apps within this chapter will be developed within this same instance of web2py.*
2. Install and activate your virtualenv.
3. Back on your command line launch web2py:

```
1 $ python web2py.py
```

4. After web2py loads, set an admin password, and you're good to go. Once you've finished your current session with web2py, exit the virtualenv:

```
1 $ deactivate
```

5. To run web2py again, activate your virtualenv and launch web2py.

NOTE web2py by default separates projects (a collection of related applications); however, it's still important to use a virtualenv 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 install 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

1. Activate your virtualenv. Fire up the server.

```
1 $ source bin/activate
2 $ python web2py.py
```

2. Input your admin password. Once logged in, click the button for the “Administrative Interface” on the right side of the page. Enter your password again. You’re now on the **Sites** page. This is the main administration page where you create and modify your applications.

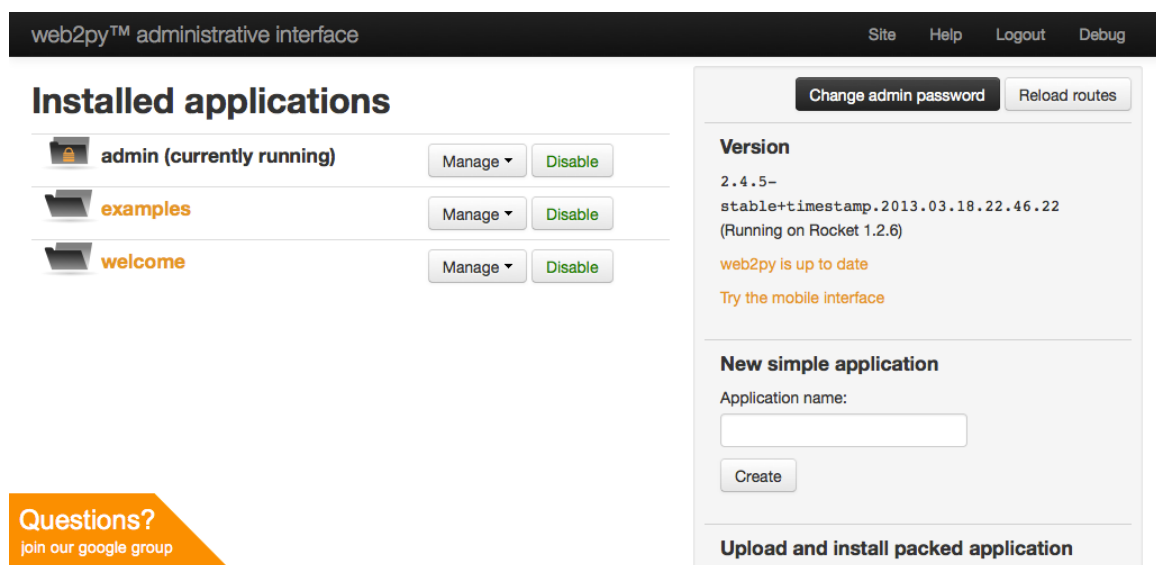


Figure 19.2: web2py Sites page

3. 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

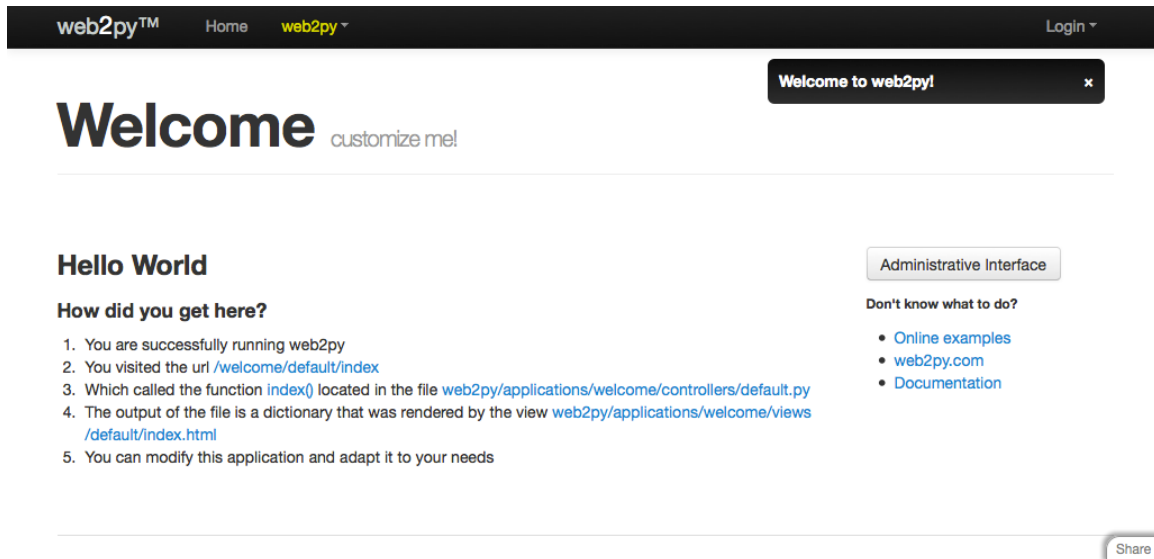


Figure 19.3: web2py Welcome

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.

4. 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

We'll go over the MVC architecture with regard to web2py in more detail in later chapters. For now, just know that it's 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.

5. Next we need to modify the default controller, *default.py*, so click the "edit" button next to the name of the file.
6. Now we're in the **web2py IDE**. Replace the `index()` function with the following code:

```
1 def index():
2     return dict(message="Hello from web2py!")
```

7. Save the file, then hit the Back button to return to the **Edit** page.
8. Now let's update the view. Edit the *default/index.html* file, replacing the existing code with:

```
1 <html>
2   <head>
3     <title>Hello App!</title>
4   </head>
5   <body>
6     <br/>
7     <h1>{{=message}}</h1>
8   </body>
9 </html>
```

9. Save the file, then return to the **Edit** page again. Click the “index” link next to the *default.py* file which loads the main 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 output 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:

For example, if we created this structure:

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`

Figure 19.4: web2py Generic View

- 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 CTRL-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.



Figure 19.5: PythonAnywhere Logo

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.

1. Go ahead and create an account and log in. Once logged in, click “Web”, then “Add a new web app”, choose `your_username.pythonanywhere.com`, and click the button for web2py. Set an admin password and then click “Next” one last time to set up the web2py project.
2. 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.

If you wanted, you could develop your entire app on PythonAnywhere. Cool, right?

3. Back on your local version of web2py return to the [admin page](#), click the “Manage” drop down button, the select “Pack All” to save the *w2p-package* to your computer.
4. 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.
5. 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 virtualenv, 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. Better yet: 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.

1. Replace the code in `default.py` with:

```
1 def index():
2     form=FORM('# of seconds: ',
3               INPUT(_name='seconds', requires=IS_NOT_EMPTY()),
4               INPUT(_type='submit')).process()
5     if form.accepted:
6         redirect(URL('convert',args=form.vars.seconds))
7     return dict(form=form)
8
9 def convert():
10    seconds = request.args(0,cast=int)
11    return dict(seconds=seconds, minutes=seconds/60,
12               new_seconds=seconds%60)
```

2. Edit the `default/index.html` view, replacing the default code with:

```
1 <center>
2 <h1>seconds2minutes</h1>
3 <h3>Please enter the number of seconds you would like converted to
   minutes.</h3>
4 <p>{{=form}}</p>
5 </center>
```

3. Create a new view called `default/convert.html`, replacing the default code with:

```

1 <center>
2 <h1>seconds2minutes</h1>
3 <p>{{=seconds}} seconds is {{=minutes}} minutes and
   {{=new_seconds}} seconds.</p>
4 <br/>
5 <p><a href="/seconds2minutes/default/index">Try again?</a><p>
6 </center>

```

4. 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.

5. Change the form validator to `requires=IS_INT_IN_RANGE(0,1000000)`. The new *index()* function looks like this:

```

1 def index():
2     form=FORM('# of seconds: ',
3               INPUT(_type='integer', _name='seconds',
4                     requires=IS_INT_IN_RANGE(0,1000000)),
5               INPUT(_type='submit')).process()
6     if form.accepted:
7         redirect(URL('convert',args=form.vars.seconds))
8     return dict(form=form)

```

Test it out. You should get an error, unless you enter an integer between 0 and 999,999:

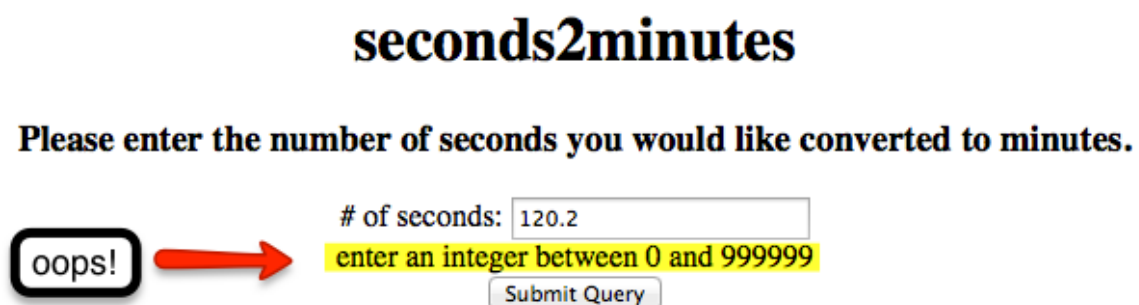


Figure 19.6: web2py error

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.

Chapter 20

Interlude: APIs

Introduction

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 request to a remote server, which responds back to the web client with the requested information.

In this chapter, we will navigate the Internet using Python programs to:

- gather data,
- access and consume web services,
- scrape web pages, and
- interact with web pages.

I'm assuming 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. 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
. *Or simply review the chapter in this course on HTML and CSS.*

These tags are used to differentiate between each section of a web site or application.

For example:

```
1 <h1>This is a headline</h1>
2 <p>This is a paragraph.</p>
3 <a href="http://www.realpython.com">This is a link.</a>
```

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 functions, which are practical in nature and which can be immediately useful to a web development project. I 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 `virtualenv`.

Install `requests`:

```
1 $ pip install requests
```

Code:

```
1 # Retrieving a web page
2
3
4 import requests
5
6 # retrieve the web page
7 r = requests.get("http://www.python.org/")
8
9 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 me show you an easier way to look at the full HTML output.

Code:

```
1 # Downloading a web page
2
3
4 import requests
5
6 r = requests.get("http://www.python.org/")
7
8 # write the content to test_request.html
9 with open("test_requests.html", "wb") as code:
10     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 Sublime. 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.

Code:

```
1 # Submitting to a web form
2
3
4 import requests
5
6 url = 'http://httpbin.org/post'
7 data = {'fname': 'Michael', 'lname': 'Herman'}
8
9 # submit post request
10 r = requests.post(url, data=data)
11
12 # display the response to screen
13 print r
```

Output:

<Response [200]>

Using the requests library, you 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:

```
1 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:

```
1 "form": {
2     "lname": "Herman",
3     "fname": "Michael"
4 },
```

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 discussed in the last chapter. 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 and web APIs, are simply APIs made available over the Internet, used for reading (GET) and writing (POST) data. GET and POST, as well as UPDATE and DELETE, along with SELECT, INSERT, UPDATE, and DELETE are all forms of CRUD:

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, WordPress reads (GETs) data from Facebook to show how many people have “liked” a blog article. Then, if you “liked” an article, data is sent (POST) to Facebook (if you allow it, of course), showing that you liked that article. Without web services, these interactions between two independent applications would be impossible.

Let's look at an example in Chrome Developer Tools:

1. Open Chrome.
2. Right click anywhere on the screen and within the window, click "Inspect Element".
3. You are now looking at the main Developer Tools pane:

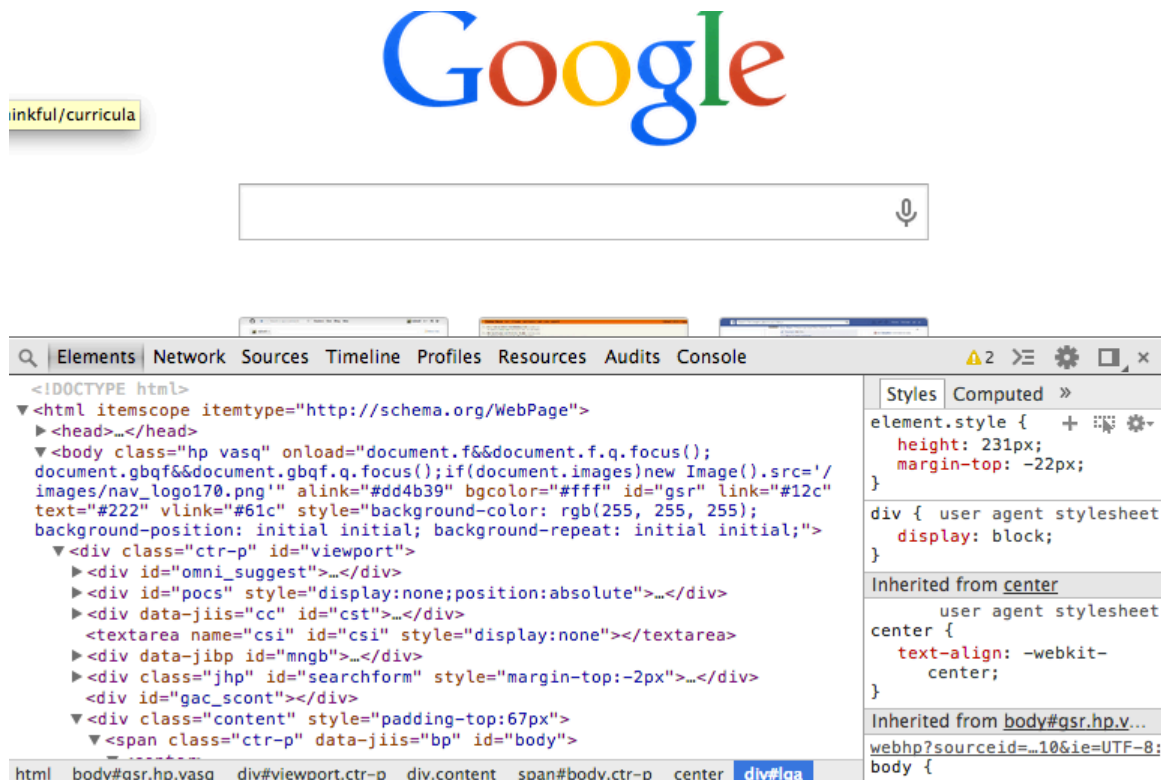


Figure 20.1: Chrome Developer Tools Example

4. Click the Network panel.
5. Now navigate in your browser to a site you need to login at. 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.
6. Now login. If you're already logged in, go ahead and log out. Enter your login credentials. Pay close attention to the console.
7. Did you see the POST request? Basically, you sent a POST request with your login credentials to the server.

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 depend on a number of factors. Access through client libraries can be easier, especially for beginners, as the code is already 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.

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 web RESTful APIs, or HTTP 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, and
- Can be access 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 rules, REST, 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 I 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 [this](#) great 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 in the next two chapters.

Before moving on, let's look at a fun example. <http://placekitten.com> is a REST 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>

Pretty cool, right?

Homework

- Read [this](#) article providing a high-level overview of standards associated with RESTful APIs.
- if you're still struggling with understanding what REST is and why we use it, [here's](#) a great video. 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.

Please note: Although, JSON continues to push XML out of the picture in terms of REST, XML is still widely used and can be easier to parse.

With that, let's look at an example of an XML file:

```
1 <?xml version="1.0"?>
2 <CARS>
3   <CAR>
4     <MAKE>Ford</MAKE>
5     <MODEL>Focus</MODEL>
6     <COST>15000</COST>
7   </CAR>
8   <CAR>
9     <MAKE>Honda</MAKE>
10    <MODEL>Civic</MODEL>
11    <COST>20000</COST>
12  </CAR>
13  <CAR>
14    <MAKE>Toyota</MAKE>
15    <MODEL>Camry</MODEL>
16    <COST>25000</COST>
17  </CAR>
18  <CAR>
19    <MAKE>Honda</MAKE>
20    <MODEL>Accord</MODEL>
21    <COST>22000</COST>
22  </CAR>
23 </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.

Code:

```
1 # XML Parsing 1
2
3
4 from xml.etree import ElementTree as et
5
6 # parses the file
7 doc = et.parse("cars.xml")
8
9 # outputs the first MODEL in the file
10 print doc.find("CAR/MODEL").text
```

Output:

```
1 Focus
```

In this program you 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:

```
1 print doc.find("CAR/MODEL").text
```

The above line is equivalent to:

```
1 print doc.find("CAR[1]/MODEL").text
```

See what happens when you change the code in the program to:

```
1 print doc.find("CAR[2]/MODEL").text
```

The output should be:

```
1 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.

Code:

```
1 # XML Parsing 2
2
3
4 from xml.etree import ElementTree as et
5
6 doc = et.parse("cars.xml")
7
8 # outputs the make, model and cost of each car to the screen
9 for element in doc.findall("CAR"):
10     print (element.find("MAKE").text + " " +
11           element.find("MODEL").text +
12           ", $" + element.find("COST").text)
```

You should get the following results:

```
1 Ford Focus, $15000
2 Honda Civic, $20000
3 Toyota Camry, $25000
4 Honda Accord, $22000
```

This program follows the same logic as the previous one, but you just added a FOR loop to iterate through the XML file, pulling all the data.

In this last example, we will use a GET request to access XML found on the web.

Code:

```
1 # XML Parsing 3
2
3
4 from xml.etree import ElementTree as et
5 import requests
6
7 # retrieve an xml document from a web server
8 xml = requests.get("http://www.w3schools.com/xml/cd_catalog.xml")
9
10 with open("test.xml", "wb") as code:
11     code.write(xml.content)
12
13 doc = et.parse("test.xml")
14
```

```
15 # outputs the album, artist and year of each CD to the screen
16 for element in doc.findall("CD"):
17     print "Album: ", element.find("TITLE").text
18     print "Artist: ", element.find("ARTIST").text
19     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:

```
1 {  
2   "CARS": [  
3     {  
4       "MAKE": "Ford",  
5       "MODEL": "Focus",  
6       "COST": "15000"  
7     },  
8     {  
9       "MAKE": "Honda",  
10      "MODEL": "Civic",  
11      "COST": "20000"  
12    },  
13    {  
14      "MAKE": "Toyota",  
15      "MODEL": "Camry",  
16      "COST": "25000"  
17    },  
18    {  
19      "MAKE": "Honda",  
20      "MODEL": "Accord",  
21      "COST": "22000"  
22    }  
23  ]  
24 }
```

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.

Without further ado, let's look at how to decode and parse a JSON file. Use the `cars.json` file.

Code:

```
1 # JSON Parsing 1
2
3
4 import json
5
6 # decodes the json file
7 output = json.load(open('cars.json'))
8
9 # display output to screen
10 print output
```

Output:

```
1 [{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:

```
1 print json.dumps(output, indent=4, sort_keys=True)
```

Your output should now look like this:

```
1 [
2     {
3         "CAR": [
```



```

4         {
5             "COST": "15000",
6             "MAKE": "Ford",
7             "MODEL": "Focus"
8         },
9         {
10            "COST": "20000",
11            "MAKE": "Honda",
12            "MODEL": "Civic"
13        },
14        {
15            "COST": "25000",
16            "MAKE": "Toyota",
17            "MODEL": "Camry"
18        },
19        {
20            "COST": "22000",
21            "MAKE": "Honda",
22            "MODEL": "Accord"
23        }
24    ]
25 }
26 ]

```

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:

```

1  # JSON Parsing 2
2
3  import json
4
5  # decodes the json file
6  output = json.load(open('cars.json'))
7
8  # display output to screen
9  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.

Code:

```
1 # POST JSON Payload
2
3
4 import json
5 import requests
6
7 url = "http://httpbin.org/post"
8 payload = {"colors": [
9     {"color": "red", "hex": "#f00"},
10    {"color": "green", "hex": "#0f0"},
11    {"color": "blue", "hex": "#00f"},
12    {"color": "cyan", "hex": "#0ff"},
13    {"color": "magenta", "hex": "#f0f"},
14    {"color": "yellow", "hex": "#ff0"},
15    {"color": "black", "hex": "#000"}
16    ]}
17 headers = {"content-type": "application/json"}
18
19 # post data to a web server
20 response = requests.post(url, data=json.dumps(payload),
21                           headers=headers)
22
23 # output response to screen
24 print response.status_code
```

Output:

```
1 200 OK
```

In some cases you will have to send data (also called 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. I have seen situations where in order to apply

for certain jobs, you have to send a Payload with your name, telephone number, and a link to your online resume.

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](#) or [Echo JSON](#) 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 examples.

Youtube

Google, the owner of YouTube, has been very liberal when it comes to providing access to their data via web APIs, allowing hundreds of thousands of developers to create their own applications.

Download the [GData](#) client library (remember what a client library is/does?) for this next example: `pip install gdata`

Code:

```
1 # Basic web services example
2
3
4 # import the client libraries
5 import gdata.youtube
6 import gdata.youtube.service
7
8 # YouTubeService() used to generate the object so that we can
   communicate with the YouTube API
9 youtube_service = gdata.youtube.service.YouTubeService()
10
11 # prompt the user to enter the Youtube User ID
12 playlist = raw_input("Please enter the user ID: ")
13
14 # setup the actual API call
15 url = "http://gdata.youtube.com/feeds/api/users/"
16 playlist_url = url + playlist + "/playlists"
17
18 # retrieve Youtube playlist
19 video_feed =
   youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)
20
21 print "\nPlaylists for " + str.format(playlist) + ":\n"
22
23 # display each playlist to screen
```

```
24 for p in video_feed.entry:
25     print p.title.text
```

Test the program out with my Youtube ID, “hermanmu”. You should see a listing of my Youtube playlists.

In the above code-

1. We started by importing the required libraries, which are for the Youtube Python client library;
2. We then established communication with Youtube; prompted the user for a user ID, and then made the API call to request the data.
3. You can see that the data was returned to the variable `video_feed`, which we looped through and pulled out certain values. Let’s take a closer look.

Comment out the loop and just output the `video_feed` variable:

```
1 # Basic web services example
2
3
4 # import the client libraries
5 import gdata.youtube
6 import gdata.youtube.service
7
8 # YouTubeService() used to generate the object so that we can
   communicate with the YouTube API
9 youtube_service = gdata.youtube.service.YouTubeService()
10
11 # prompt the user to enter the Youtube User ID
12 playlist = raw_input("Please enter the user ID: ")
13
14 # setup the actual API call
15 url = "http://gdata.youtube.com/feeds/api/users/"
16 playlist_url = url + playlist + "/playlists"
17
18 # retrieve Youtube playlist
19 video_feed =
   youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)
20
```

```

21 print video_feed
22
23 # print "\nPlaylists for " + str.format(playlist) + ":\n"
24
25 # # display each playlist to screen
26 # for p in video_feed.entry:
27 #     print p.title.text

```

Look familiar? You should be looking at an XML file. It's difficult to read, though. So, how did I know which elements to extract? Well, I went and looked at the [API documentation](#).

Navigate to that URL.

First, you can see the URL for connecting with (calling) the API and extracting a user's information with regard to playlists:

<http://gdata.youtube.com/feeds/api/users/username/playlists>

Try replacing username with my username, hermanmu, then navigate to this URL in your browser. You should see the same XML file that you did just a second ago when you printed just the `video_feed` variable.

Again, this is such a mess we can't read it. At this point, we could download the file, like we did in lesson 2.2, to examine it. But first, let's look at the documentation some more. Perhaps there's a clue there.

Scroll down to the "Retrieving playlist information", and look at the code block. You can see that there is sample code there for iterating through the XML file:

```

1 # iterate through the feed as you would with any other
2 for playlist_video_entry in playlist_video_feed.entry:
3     print playlist_video_entry.title.text

```

With enough experience, you will be able to just look at this and know that all you need to do is append `title` and `text`, which is what I did in the original code to obtain the required information:

```

1 for p in video_feed.entry:
2     print p.title.text

```

For now, while you're still learning, you have one of two options:

1. Trial and error, or
2. Download the XML file using the requests library and examine the contents

Always look at the documentation first. You will usually find something to work with, and again this is how you learn. Then if you get stuck, use the “Google-it-first” algorithm/philosophy, then if you still have trouble, go ahead and download the XML.

By the way, one of my readers helped with developing the following code to pull not only the title of the playlist but the list of videos associated with each playlist as well. Cheers!

```
1 # Basic web services example
2
3 # import the client libraries
4 import gdata.youtube
5 import gdata.youtube.service
6
7 # YouTubeService() used to generate the object so that we can
   communicate with the YouTube API
8 youtube_service = gdata.youtube.service.YouTubeService()
9
10 # prompt the user to enter the Youtube User ID
11 user_id = raw_input("Please enter the user ID: ")
12
13 # setup the actual API call
14 url = "http://gdata.youtube.com/feeds/api/users/"
15 playlist_url = url + user_id + "/playlists"
16
17 # retrieve Youtube playlist and video list
18 playlist_feed =
   youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)
19 print "\nPlaylists for " + str.format(user_id) + ":\n"
20
21 # display each playlist to screen
22 for playlist in playlist_feed.entry:
23     print playlist.title.text
24     playlistid = playlist.id.text.split('/')[1]
25     video_feed =
       youtube_service.GetYouTubePlaylistVideoFeed(playlist_id =
       playlistid)
26     for video in video_feed.entry:
27         print "\t"+video.title.text
```

Does this make sense? Notice the nested loops. What’s different about this code from the previous code?

Twitter

Like Google, Twitter provides a very open API. I use the Twitter API extensively for pulling in tweets on specific topics, then parsing and extracting them to a CSV file for analysis. One of the best client libraries to use with the Twitter API is [Tweepy](#): `pip install tweepy`.

Before we look at example, you need to obtain access codes for authentication.

1. Navigate to <https://dev.twitter.com/apps>.
2. Click the button to create a new application.
3. Enter dummy data.
4. After you register, you will be taken to your application where you will need the following access codes:
 - consumer_key
 - consumer_secret
 - access_token
 - access_secret
5. Make sure to create your access token to obtain the access_token and access_secret.

Now, let's look at an example:

```
1 # Twitter Web Services
2
3
4 import tweepy
5
6 consumer_key    = "<get_your_own>"
7 consumer_secret = "<get_your_own>"
8 access_token    = "<get_your_own>"
9 access_secret   = "<get_your_own>"
10
11 auth = tweepy.auth.OAuthHandler(consumer_key, consumer_secret)
12 auth.set_access_token(access_token, access_secret)
13 api = tweepy.API(auth)
14
15 tweets = api.search(q='#python')
```



```

16
17 # display results to screen
18 for t in tweets:
19     print t.created_at, t.text, "\n"

```

NOTE Add your keys and tokens in the above code before running.

If done correctly, this should output the tweets and the dates and times they were created:

```

1 2014-01-26 16:26:31 RT @arialdomartini: #pymacs is a crazy tool
   that allows the use of #python as external language for
   programming and extending #emacs http...:/
2
3 2014-01-26 16:22:43 The Rise And Fall of Languages in 2013
   http://t.co/KN4gaJASkn via @dr_dobbs #Python #Perl #C++
4
5 2014-01-26 16:21:28 #pymacs is a crazy tool that allows the use of
   #python as external language for programming and extending
   #emacs http://t.co/m2oU1ApDJp
6
7 2014-01-26 16:20:06 RT @PyCero91: Nuevo #MOOC muy interesante de
   programación con #Python de la Universidad de #Rice
   http://t.co/OXjiT38Wk0
8
9 2014-01-26 16:19:50 nice to start to email list only early bird
   sales for snakes on a train! http://t.co/A7bemXStDt #pycon
   #python
10
11 2014-01-26 16:15:31 RT @sarahkendrew: this is super #python &gt; RT
   @davidwhogg: In which @jakevdp extolls #hackAAS and tells us
   about tooltips http://t.co/...s47SkX
12
13 2014-01-26 16:14:15 this is super #python &gt; RT @davidwhogg: In
   which @jakevdp extolls #hackAAS and tells us about tooltips
   http://t.co/s47SkXw9wa
14
15 2014-01-26 16:12:00 Most of this stuff doesn't bother me. Means
   that I should use Python more pycoders: What I Hate About
   Python http://t.co/3J4yScMLpE #python
16

```

```

17 2014-01-26 16:06:15 analog - analog - Log Analysis Utility pypi:
    http://t.co/aDakEdKQ4j www: https://t.co/1rRCffeH90 #python
18
19 2014-01-26 16:05:52 RT @pycoders: Parallelism in one line
    http://t.co/XuVjYXEiJ3 #python
20
21 2014-01-26 16:00:04 RT @a_bhi_9: A 100 step for loop being iterated
    100000000 times for 4 different possibilities. I am pulling my
    7 year old to the limit I ...be
22
23 2014-01-26 16:00:04 RT @a_bhi_9: Timing analysis of #iterators
    shows NoneType iterators perform way more better than list
    #generators. #Python #analysis
24
25 2014-01-26 15:54:07 #python #table #dive http://t.co/4wHxDgJCE5
    Dive Into Python
26
27 2014-01-26 15:51:03 RT @takenji_ebooks: I think I'm going to write
    in #Python 3 as much as possible rather than 2 from now on.
28
29 2014-01-26 15:49:04 RT @pycoders: What I Hate About Python
    http://t.co/URI1gh0sqA #python

```

Essentially, the search results were returned to a list, and then you used a For loop to iterate through that list to extract the desired information.

How did I know I wanted the keys created at and text? Again, trial and error. I started with the [documentation](#), then I did some Google searches on my own. You will 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!

Please Note:

Make sure you change your keys and token variables back to-

```

1 consumer_key      = "<get_your_own>"
2 consumer_secret   = "<get_your_own>"
3 access_token      = "<get_your_own>"
4 access_secret     = "<get_your_own>"

```

-before you PUSH to Github. *You do not want anyone else but you to know those keys.*

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..

1. Use the following URL to call (or invoke) the API:

```
1 https://maps.googleapis.com/maps/  
2   api/directions/output?parameters
```

2. You then must specify the output. We'll use JSON since it's easier to work with.
3. Also, you must specify some parameters. Notice in the documentation how some parameters are required while others are optional. Let's use these parameters:

```
1 origin=Central+Park  
2 destination=Times+Square  
3 sensor=false  
4 mode=walking
```

4. You could simply append the output as well as the parameters to the end of the URL -

```
1 https://maps.googleapis.com/maps/api/directions/json?origin=  
2   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:

All right. Let's breakdown the for loops:

```
1 for route in output['routes']:  
2     for leg in route['legs']:  
3         for step in leg['steps']:  
4             print step['html_instructions']
```

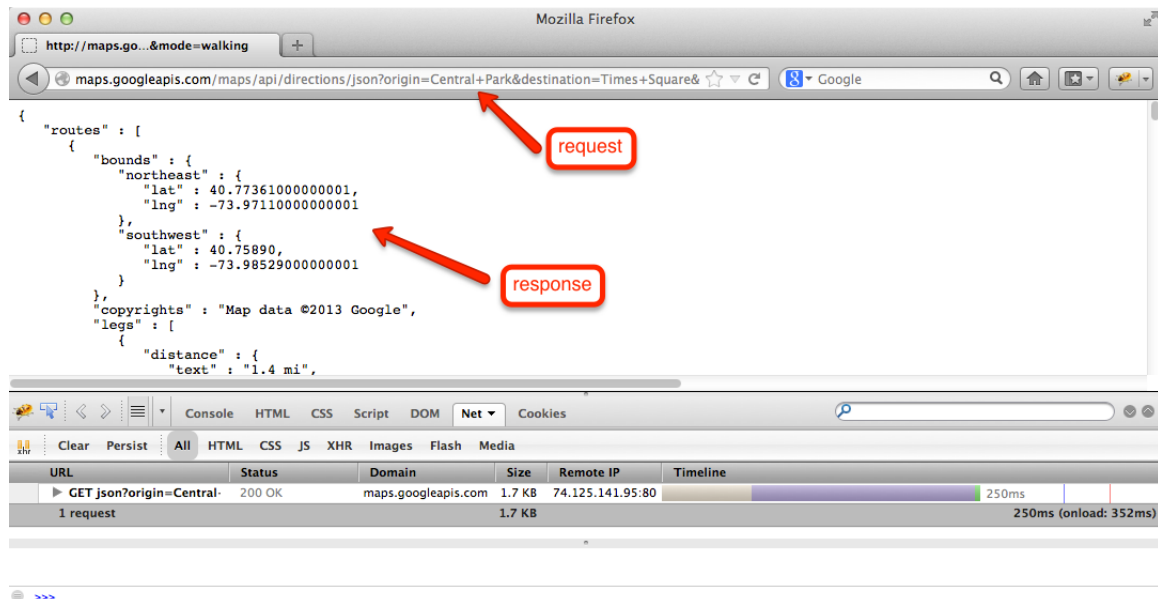


Figure 20.2: Google Directions API Request/Response



Figure 20.3: Google Directions API - parsing the response

```

{
  "routes": [
    {
      "bounds": {
        "northeast": {
          "lat": 40.773610000000001,
          "lng": -73.971100000000001
        },
        "southwest": {
          "lat": 40.75890,
          "lng": -73.985290000000001
        }
      },
      "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": -73.985290000000001
          },
          "start_address": "Central Park, 14 East 60th Street, New York, NY 10022, USA",
          "start_location": {
            "lat": 40.773610000000001,
            "lng": -73.971100000000001
          },
          "steps": [
            {
              "distance": {
                "text": "0.2 mi",
                "value": 328
              },
              "duration": {
                "text": "4 mins",
                "value": 241
              },
              "end_location": {
                "lat": 40.770790000000001,
                "lng": -73.97220
              },
              "html_instructions": "Head \u003cb\u003esouth\u003c/b\u003e on \u003cb\u003eThe Mall\u003c/b\u003e",

```

Figure 20.4: Google Directions API - parsing the response (continued)

Compare the loops to the entire output. You can see that for each loop we're just moving in (or down) one level:

So, if I wanted to print the `start_address` and `end_address`, I would just need two for loops:

```
1 for route in output['routes']:
2     for leg in route['legs']:
3         print leg['start_address']
4         print leg['end_address']
```

Homework

- Using the Google Direction API, pull driving directions from San Francisco to Los Angeles in XML. Extract the step-by-step driving 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, always, ALWAYS* want to start with the documentation. Again, all APIs work a little differently because few universal standards or practices have been established. Fortunately, the My API Films is not only well documented - but also easy to read and follow.

In this example, we want to grab a list of all movies currently playing in theaters, which we can grab from this endpoint - <http://api.myapifilms.com/imdb.do#inTheatersExample>. Try it!

Endpoints are the actual connection points for accessing the data. In other words, they are the specific URLs used for calling an API to GET data. Each endpoint is generally associated with a different type of data, which is why endpoints are often associated with groupings of data (e.g., movies playing in the theater, movies opening on a certain date, top rentals, and so on).

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. It's less important that you understand how each method works than to understand *how* to obtain authentication to access the web service. Always refer to the web service provider's documentation to obtain this information.

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 either 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

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, parsing the relevant data, and then dumping it into a database. We'll do this in iterations.

Let's start by writing a script to create the database and table.

Code:

```
1 # Create a SQLite3 database and table
2
3
4 import sqlite3
5
6 with sqlite3.connect("movies.db") as connection:
7     c = connection.cursor()
8
9     # create a table
10    c.execute("""CREATE TABLE new_movies
11                (title TEXT, year INT, votes text,
12                release_date text, rating INT, metascore
13                INT)""")
```

Save this as *create_db.py* and then run it:

```
1 $ python create_db.py
```

We need one more script now to pull the data and dump it directly to the database:

```
1 # GET data from My API Films, parse, and write to database
2
3
4 import json
5 import requests
6 import sqlite3
7
8
9 TOKEN = 'b6631079-25cd-4319-ae07-c7279cabcc12'
10 url =
11     requests.get('http://api.myapifilms.com/imdb/inTheaters?token={0}&format=json&l
12
13 # convert data from feed to binary
14 binary = url.content
15
16 # decode the json feed
```



```

16 output = json.loads(binary)
17
18 # grab the list of movies
19 movies = output['data']['inTheaters']
20
21 with sqlite3.connect("movies.db") as connection:
22     c = connection.cursor()
23
24     # iterate through each movie and write to the database
25     for movie in movies:
26         all_movies = movie['movies']
27         for meta in all_movies:
28             if(meta['title']):
29                 c.execute("INSERT INTO new_movies VALUES(?, ?, ?,
30                             ?, ?, ?)",
31                             (meta["title"], meta["year"],
32                             meta["votes"],
33                             meta["releaseDate"], meta["metascore"],
34                             meta["rating"]))
35
36     # retrieve data
37     c.execute("SELECT * FROM new_movies ORDER BY title ASC")
38
39     # fetchall() retrieves all records from the query
40     rows = c.fetchall()
41
42     # output the rows to the screen, row by row
43     for r in rows:
44         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?

1. We grabbed the endpoint URL with a GET request.
2. Converted the data to binary.
3. Decoded the JSON feed.

4. Then used a for loop to write the data to the database.
5. Finally we grabbed 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!*

Chapter 21

web2py: Sentiment Analysis

Sentiment Analysis

Continue to use the same web2py instance as before for the apps in this chapter as well.

What is Sentiment Analysis?

Essentially, [sentiment analysis](#) measures the sentiment of something - a feeling rather than a fact, in other words. 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 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, please read this amazing [article](#).

1. 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. If you can't answer them by the end of this lesson, try the "Google-it-first" method, then, if you still have questions, post them to the Real Python [message forum](#).
2. 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.

Traditionally, you would access cURL from the terminal in Unix systems. Unfortunately, for Windows users, command prompt 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.

3. Unix users, and Windows users with Cygwin installed, test out the API in the terminal:

```
1 $ curl -d "text=great" http://text-processing.com/api/sentiment/  
2 {"probability": {"neg": 0.35968353095023886, "neutral":  
   0.29896828324578045, "pos": 0.64031646904976114}, "label":  
   "pos"}  
3  
4 $ curl -d "text=i hate apples"  
   http://text-processing.com/api/sentiment/  
5 {"probability": {"neg": 0.65605365432549356, "neutral":  
   0.3611947857779943, "pos": 0.34394634567450649}, "label": "neg"}  
6  
7 $ curl -d "text=i usually like ice cream but this place is  
   terrible" http://text-processing.com/api/sentiment/  
8 {"probability": {"neg": 0.90030914036608489, "neutral":  
   0.010418429982506104, "pos": 0.099690859633915108}, "label":  
   "neg"}  
9  
10 $ curl -d "text=i really really like you, but today you just  
    smell." http://text-processing.com/api/sentiment/  
11 {"probability": {"neg": 0.638029187699517, "neutral":  
    0.001701536649255885, "pos": 0.36197081230048306}, "label":  
    "neg"}
```

4. 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?

5. You can also test the API on [Hurl](#):

A screenshot of the Hurl web interface. At the top is the URL input field containing 'http://text-processing.com/api/sentiment/'. Below it is a dropdown menu set to 'POST' and a checkbox for 'follow redirects'. There are links for '+ add param' and '+ set post body'. A parameter named 'text' is added with a value of '"i greatly dislike perl"'. Below this are radio buttons for 'no auth' (selected) and 'HTTP basic', along with a '+ add header' link. A 'Send' button is at the bottom.

Figure 21.1: Hurl POST Request

Steps:

- Enter the URL
- Change the HTTP method to POST
- Add the parameter text and `"i greatly dislike perl"`
- Press send

Surprised at the results?

Requests

1. 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:

```
1 import requests
2
3 url = 'http://text-processing.com/api/sentiment/'
4 data = {'text': 'great'}
```

```

5 r = requests.post(url, data=data)
6 print r.content

```

Go ahead and install the requests library. Wait. Didn't we already do that? *Remember: Since we're in a different virtualenv, we need to install it again.*

Remember the command `pip install requests`?

Now, test out the above code in your Shell:

```

(web2py)Michaels-MacBook-Pro:web2py michaelherman$ python
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[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
{"probability": {"neg": 0.35968353095023886, "neutral": 0.29896828324578045, "pos": 0.64031646904976114}, "label": "pos"}
>>> print r
<Response [200]>
>>> 

```

Figure 21.2: Sentiment API Response

Now, let's build the app for easily testing sentiment analysis. It's called Pulse.

Pulse

1. You know the drill: Activate your virtualenv, fire up the server, enter the Admin Interface, and create a new app called "pulse".
2. 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`.
3. Replace the code in the default controller with:

```

1 import requests
2
3 def index():
4     form=FORM(
5         TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
6         INPUT(_type='submit')).process()
7     if form.accepted:
8         redirect(URL('pulser',args=form.vars.pulse))
9     return dict(form=form)
10
11 def pulser():
12     text = request.args(0)
13     text = text.split('_')
14     text = ' '.join(text)
15     url = 'http://text-processing.com/api/sentiment/'
16     data = {'text': text}
17     r = requests.post(url, data=data)
18     return dict(text=text, r=r.content)

```

Now let's create some basic views.

4. *default/index.html*:

```

1 {{extend 'layout.html'}}
2 <center>
3 <br/>
4 <br/>
5 <h1>check a pulse</h1>
6 <h4>Just another Sentiment Analysis tool.</h4>
7 <br/>
8 <p>{{=form}}</p>
9 </center>

```

5. *default/pulser.html*:

```

1 {{extend 'layout.html'}}
2 <center>
3 <p>{{=text}}</p>
4 <br/>
5 <p>{{=r}}</p>
6 <br/>
7 <p><a href="/pulse/default/index">Another Pulse?</a><p>

```

```
8 </center>
```

6. Alright, 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:

```
1 $ pip install requests
```

Test it again. Did it work this time?

7. Now, let's finish cleaning up *pulser.html*. We need to parse/decode 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? That would be cool. It all depends on your (intended) audience. Let's just parse out the end result.

8. Update *default.py*:

```
1 import requests
2 import json
3
4 def index():
5     form=FORM(
6         TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
7         INPUT(_type='submit')).process()
8     if form.accepted:
9         redirect(URL('pulser',args=form.vars.pulse))
10    return dict(form=form)
11
12 def pulser():
13     text = request.args(0)
14     text = text.split('_')
15     text = ' '.join(text)
16
17     url = 'http://text-processing.com/api/sentiment/'
18     data = {'text': text}
19
20     r = requests.post(url, data=data)
21
22     binary = r.content
23     output = json.loads(binary)
24     label = output["label"]
```



```

25
26     return dict(text=text, label=label)

```

9. Update *default/pulser.html*:

```

1  {{extend 'layout.html'}}
2  <center>
3  <br/>
4  <br/>
5  <h1>your pulse</h1>
6  <h4>{{=text}}</h4>
7  <p>is</p>
8  <h4>{{=label}}</h4>
9  <br/>
10 <p><a href="/pulse/default/index">Another Pulse?</a><p>
11 </center>

```

10. Make it pretty. Update *layout.html*:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>check your pulse</title>
5  <meta charset="utf-8" />
6  <style type="text/css">
7      body {font-family: Arial, Helvetica, sans-serif;
8           font-size:x-large;}
9  </style>
10  {{
11  middle_columns = {0:'span12',1:'span9',2:'span6'}
12  }}
13  {{block head}}{{end}}
14 </head>
15 <body>
16   <div class="{{=middle_columns}}">
17       {{block center}}
18       {{include}}
19       {{end}}
20   </div>
21 </body>
</html>

```

11. Test it out. Is it pretty? No. It's functional:

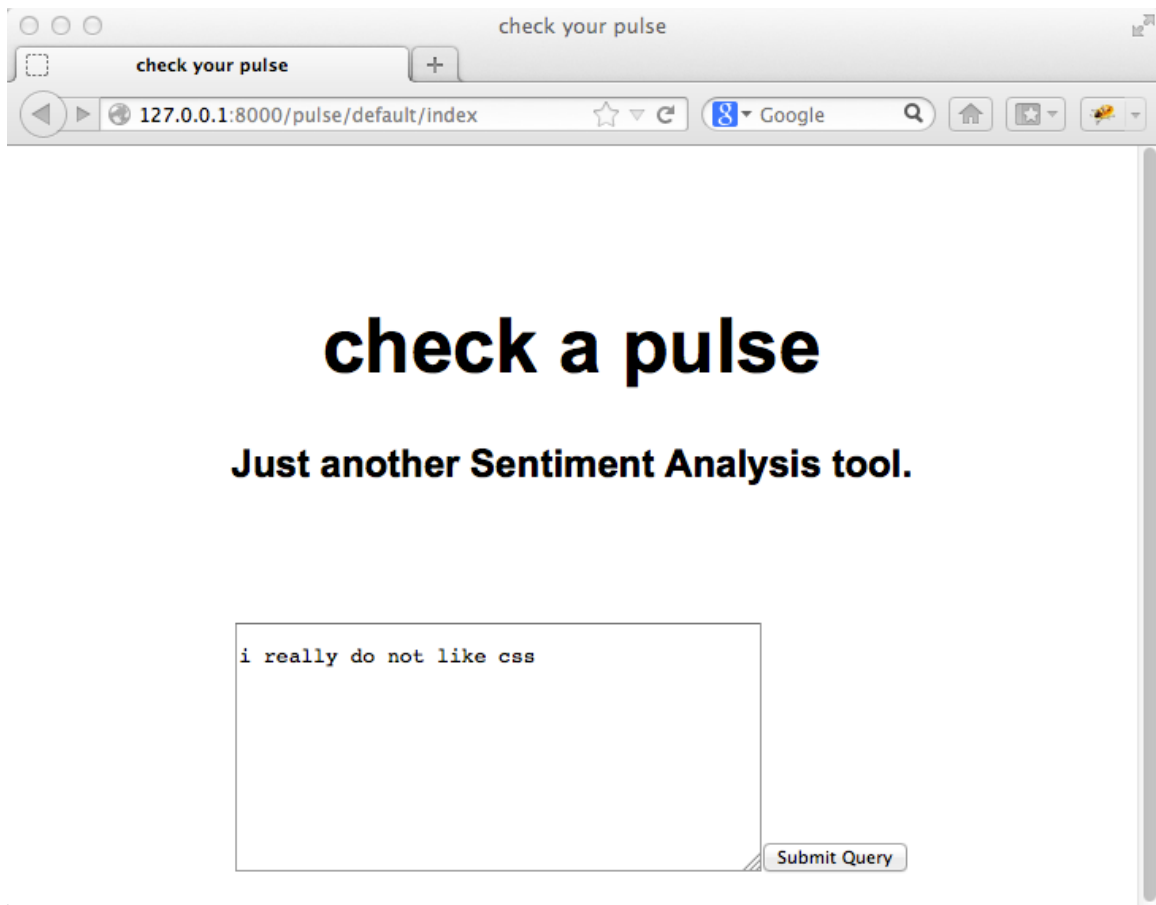


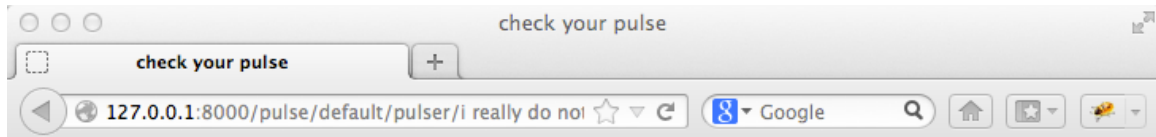
Figure 21.3: Pulse App Example 1

12. Make yours pretty.

13. Better?

Next steps

What else could you do with this? Well, you could easily tie a database into 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?



your pulse

i really do not like css

is

neg

[Another Pulse?](#)

Figure 21.4: Pulse App Example 2

check a pulse

Just another Sentiment Analysis tool.

yay|

Submit

Figure 21.5: Pulse App Example 3

your pulse

yay
is
neutral

Another Pulse?

Figure 21.6: Pulse App Example 4

```
{"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 [this](#) 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.

NOTE: If you don't know what Javascript, jQuery, or AJAX is, take a look at [this](#) StackOverflow article. Still confused? Google it. We will cover Javascript and jQuery in a latter chapter. Feel free to jump ahead.

web2py and AJAX

1. 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.

```
1 {{extend 'layout.html'}}
2 <h1>AJAX Test</h1>
3 <form>
4     <input type="number" id="key" name="key">
5     <input type="button" value="submit"
6         onclick="ajax('{{=URL('data')}}', ['key'], 'target')">
7 </form>
8 <br>
9 <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 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`.

2. Now update the following code to the new controller:

```
1 def index():
2     return dict()
3
4 def data():
5     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.

3. Now let's update the parent template *layout.html*:

```
1 <!DOCTYPE html>
2 <head>
3     <title>AJAX Test</title>
4     <script
5         src="{%=URL('static','js/modernizr.custom.js')%}"></script>
6     <!-- include stylesheets -->
7     {{
8         response.files.insert(0,URL('static','css/web2py.css'))
9         response.files.insert(1,URL('static','css/bootstrap.min.css'))
10        response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
11        response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
12    }}
13    {{include 'web2py_ajax.html'}}
14    {{
15        middle_columns = {0:'span12',1:'span9',2:'span6'}
16    }}
17    <noscript><link href="{%=URL('static',
18        'css/web2py_bootstrap_nojs.css')%}" rel="stylesheet"
19        type="text/css" /></noscript>
20    {{block head}}{{end}}
21 </head>
22 <body>
23     <div class="container">
24         <section id="main" class="main row">
25             <div class="{%=middle_columns%}">
26                 {{block center}}
27                 {{include}}
28                 {{end}}
29             </div>
```



```

27     </section><!--/main-->
28 </div> <!-- /container -->
29 <script src="{%=URL('static','js/bootstrap.min.js')%}"></script>
30 <script
    src="{%=URL('static','js/web2py_bootstrap.js')%}"></script>
31 </body>
32 </html>

```

4. Test it out. Make sure to enter an integer. You should see this (if you entered 15, of course):

AJAX Test

25

Figure 21.7: Pulse App Example 5

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 behavior of the page.

Let's apply this to our Pulse app.

Adding AJAX to Pulse

1. Update the controller:

```
1 import requests
2
3 def index():
4     return dict()
5
6 def pulse():
7     session.m=[]
8     if request.vars.sentiment:
9         text = request.vars.sentiment
10        text = text.split('_')
11        text = ' '.join(text)
12        url = 'http://text-processing.com/api/sentiment/'
13        data = {'text': text}
14        r = requests.post(url, data=data)
15        session.m.append(r.content)
16        session.m.sort()
17        return text, TABLE(*[TR(v) for v in session.m]).xml()
```

Here, we're simply grabbing the text from the input, running the analysis, appending the results to a list, and then returning the original text along with the results.

2. Update the view, *default/index.html*:

```
1 {{extend 'layout.html'}}
2 <h1>check your pulse</h1>
3 <form>
4     <input type="text" id="sentiment" name="sentiment">
5     <input type="button" value="submit"
6         onclick="ajax('{{URL('pulse')}}', ['sentiment'], 'target')">
7 </form>
8 <br>
9 <div id="target"></div>
```

Nothing new here.

3. Test it out. You should see something like:

check your pulse

web2py

```
{"probability": {"neg": 0.50955199890675162, "neutral": 0.59508906140405482, "pos": 0.49044800109324838}, "label": "neutral"}
```

Figure 21.8: Pulse App Example 6

Additional Features

Now, let's expand our app so that we can enter two text inputs for comparison.

1. Update the controller:

```
1 import requests
2
3 def index():
4     return dict()
5
6 def pulse():
7
8     session.m=[]
9     url = 'http://text-processing.com/api/sentiment/'
10
11     # first item
12     text_first = request.vars.first_item
13     text_first = text_first.split('_')
14     text_first = ' '.join(text_first)
15     data_first = {'text': text_first}
16     r_first = requests.post(url, data=data_first)
17     session.m.append(r_first.content)
18
19     # second_item
20     text_second = request.vars.second_item
21     text_second = text_second.split('_')
```

```

22 text_second = ' '.join(text_second)
23 data_second = {'text': text_second}
24 r_second = requests.post(url, data=data_second)
25 session.m.append(r_second.content)
26
27 session.m.sort()
28 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.

2. Update the view:

```

1 {{extend 'layout.html'}}
2 <h1>pulse</h1>
3 <p>comparisons using sentiment</p>
4 <br>
5 <form>
6   <input type="text" id="first_item" name ="first_item"
   placeholder="enter first item...">
7   <br>
8   <input type="text" id="second_item" name ="second_item"
   placeholder="enter second item...">
9   <br>
10  <input type="button" value="submit"
   onclick="ajax('{{URL('pulse')}}', ['first_item', 'second_item'], 'target')
11 </form>
12
13 <br>
14 <div id="target"></div>

```

Notice how we're passing two items to the pulse URL, ['first_item', 'second_item'].

3. Test.

Now, let's make this look a little nicer.

4. Only display the item that has the higher (more positive) sentiment. To do this update the controller:

```

1 import requests
2 import json
3

```

pulse

comparisons using sentiment

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"}
```

Figure 21.9: Pulse App Example 7

```
4 def index():  
5     return dict()  
6  
7 def process(my_list):  
8  
9     # analyze first item  
10    binary_first = my_list[1]  
11    output_first = json.loads(binary_first)  
12    label_first = output_first["label"]  
13  
14    # analyze second item  
15    binary_second = my_list[3]  
16    output_second = json.loads(binary_second)  
17    label_second = output_second["label"]  
18  
19    # logic  
20    if label_first == "pos":  
21        if label_second != "pos":  
22            return my_list[0]  
23        else:  
24            if output_first["probability"]["pos"] >  
25                output_second["probability"]["pos"]:  
26                return my_list[0]  
27            else:
```

```

27         return my_list[2]
28 elif label_first == "neg":
29     if label_second != "neg":
30         return my_list[2]
31     else:
32         if output_first["probability"]["neg"] <
33             output_second["probability"]["neg"]:
34             return my_list[0]
35         else:
36             return my_list[2]
37 elif label_first == "neutral":
38     if label_second == "pos":
39         return my_list[2]
40     elif label_second == "neg":
41         return my_list[0]
42     else:
43         if output_first["probability"]["pos"] >
44             output_second["probability"]["pos"]:
45             return my_list[0]
46         else:
47             return my_list[2]
48
49 def pulse():
50
51     session.m=[]
52     url = 'http://text-processing.com/api/sentiment/'
53
54     # first item
55     text_first = request.vars.first_item
56     text_first = text_first.split('_')
57     text_first = ' '.join(text_first)
58     session.m.append(text_first)
59     data_first = {'text': text_first}
60     r_first = requests.post(url, data=data_first)
61     session.m.append(r_first.content)
62
63     # second item
64     text_second = request.vars.second_item
65     text_second = text_second.split('_')
66     text_second = ' '.join(text_second)

```

```

65     session.m.append(text_second)
66     data_second = {'text': text_second}
67     r_second = requests.post(url, data=data_second)
68     session.m.append(r_second.content)
69
70     winner = process(session.m)
71
72     return "The winner is {}".format(winner)

```

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.

1. First update the layout template with a bootstrap stylesheet as well as some custom styles:

```

1 <!DOCTYPE html>
2 <head>
3     <title>AJAX Test</title>
4     <script
5         src="{URL('static','js/modernizr.custom.js')}"></script>
6     <!-- include stylesheets -->
7     {{
8         response.files.insert(0,URL('static','css/web2py.css'))
9         response.files.insert(1,URL('static','css/bootstrap.min.css'))
10        response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
11        response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
12    }}
13    {{include 'web2py_ajax.html'}}
14    {{
15        middle_columns = {0:'span12',1:'span9',2:'span6'}
16    }}
17    <link href="http://maxcdn.bootstrapcdn.com/
18        bootswatch/3.2.0/yeti/bootstrap.min.css" rel="stylesheet">
19    <noscript><link href="{URL('static',
20        'css/web2py_bootstrap_nojs.css')}" rel="stylesheet"
21        type="text/css" /></noscript>

```

```

19     {{block head}}{{end}}
20 </head>
21 <body>
22     <div class="container">
23         <div class="jumbotron">
24             <div class="{{=middle_columns}}">
25                 {{block center}}
26                 {{include}}
27                 {{end}}
28             </div>
29         </div>
30     </div> <!-- /container -->
31     <script src="{{=URL('static','js/bootstrap.min.js')}}"></script>
32     <script
33         src="{{=URL('static','js/web2py_bootstrap.js')}}"></script>
34 </body>
</html>

```

2. Now update the child view:

```

1 {{extend 'layout.html'}}
2 <h1>pulse</h1>
3 <p class="lead">comparisons using sentiment</p>
4 <br>
5 <form role="form">
6     <div class="form-group">
7         <input type="text" id="first_item" class="form-control"
8             name="first_item" placeholder="enter first item...">
9         <br>
10        <input type="text" id="second_item" class="form-control"
11            name="second_item" placeholder="enter second item...">
12        <br>
13        <input type="button" class="btn btn-success" value="submit"
14            onclick="ajax('{{=URL('pulse')}}',
15                ['first_item','second_item'],'target')">
16    </div>
17 </form>
18 <br>
19 <h2 id="target"></h2>

```


We simply added some bootstrap classes to make it look much nicer.
Looking good!

pulse

comparisons using sentiment

The winner is python!

Figure 21.10: Pulse App Example 8 - python vs. ruby!

Movie Suggester

Let's take this to the next level and create a movie suggester. Essentially, we'll pull data from the Rotten Tomatoes API to grab movies that are playing then display the sentiment of each.

Setup

1. Create a new app called "movie_suggest".
2. Replace the code in the controller, *default.py* with:

```
1 import requests
2 import json
3
4 def index():
5     return dict()
6
7 def grab_movies():
8     session.m = []
9     YOUR_OWN_KEY = 'GET_YOUR_OWN_KEY'
10    url =
11        requests.get("http://api.rottentomatoes.com/api/public/v1.0/"
12            +
13            "lists/movies/in_theaters.json?apikey={}".format(YOUR_OWN_KEY))
14    binary = url.content
15    output = json.loads(binary)
16    movies = output['movies']
17    for movie in movies:
18        session.m.append(movie["title"])
19    session.m.sort()
20    return TABLE(*[TR(v) for v in session.m]).xml()
```

NOTE: Make sure you add your API key into the value for the variable `YOUR_OWN_KEY`. Forgot your key? Return to the last chapter to find out how to obtain a new one.

In this script, we're using the Rotten Tomatoes API to grab the movies currently playing.

3. Test this out in your shell to see exactly what's happening:

```

1 >>> import requests
2 >>> import json
3 >>> YOUR_OWN_KEY = 'abha3gx3p42czdrswkejmyxm'
4 >>> url =
    requests.get("http://api.rottentomatoes.com/api/public/v1.0/" +
5 ...
    "lists/movies/in_theaters.json?apikey={}".format(YOUR_OWN_KEY))
6 >>> binary = url.content
7 >>> output = json.loads(binary)
8 >>> movies = output['movies']
9 >>> for movie in movies:
10 ...     movie["title"]
11 ...
12 u'The Lego Movie'
13 u'Pompeii'
14 u'Son Of God'
15 u'Non-Stop'
16 u'3 Days To Kill'
17 u'The Monuments Men'
18 u'RoboCop'
19 u'Ride Along'
20 u'About Last Night'
21 u'Endless Love'
22 u'Anchorman 2: The Legend Continues'
23 u'Lone Survivor'
24 u'American Hustle'
25 u'The Nut Job'
26 u'Winter's Tale'
27 u'The Wind Rises'

```

4. Update the view, *default/index.html*:

```

1 {{extend 'layout.html'}}
2 <h1>suggest-a-movie</h1>
3 <p>use sentiment to find that perfect movie</p>
4 <br>
5 <form>
6     <input type="button" value="Get Movies"
7         onclick="ajax('{{URL('grab_movies')}}', [], 'target')">
8 </form>

```

```

8 <br>
9 <div id="target"></div>

```

5. Update the parent view, *layout.html*:

```

1 <!DOCTYPE html>
2 <head>
3 <title>suggest-a-movie</title>
4 <script src="{%=URL('static','js/modernizr.custom.js')%}"></script>
5 <!-- include stylesheets -->
6 {{
7 response.files.insert(0,URL('static','css/web2py.css'))
8 response.files.insert(1,URL('static','css/bootstrap.min.css'))
9 response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
10 response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
11 }}
12 {{include 'web2py_ajax.html'}}
13 {{
14 middle_columns = {0:'span12',1:'span9',2:'span6'}
15 }}
16 <link href="http://maxcdn.bootstrapcdn.com/
17     bootswatch/3.2.0/yeti/bootstrap.min.css" rel="stylesheet">
18 <noscript><link href="{%=URL('static',
19     'css/web2py_bootstrap_nojs.css')%}" rel="stylesheet"
20     type="text/css" /></noscript>
21 {{block head}}{{end}}
22 </head>
23 <body>
24 <div class="container">
25     <div class="jumbotron">
26         <div class="{%=middle_columns%}">
27             {{block center}}
28             {{include}}
29             {{end}}
30         </div>
31     </div>
32 </div> <!-- /container -->
33 <script src="{%=URL('static','js/bootstrap.min.js')%}"></script>
34 <script src="{%=URL('static','js/web2py_bootstrap.js')%}"></script>
35 </body>

```

34 `</html>`

6. 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

Figure 21.11: Suggest-a-Movie Example 1

Add Sentiment

Let's now determine the sentiment of each movie.

1. Update the controller:

```
1 import requests
2 import json
3
4 def index():
5     return dict()
6
7 def grab_movies():
8     session.m=[]
9     YOUR_OWN_KEY = 'abha3gx3p42czdrswkejmyxm'
10    url =
11        requests.get("http://api.rottentomatoes.com/api/public/v1.0/"
12        +
13        "lists/movies/in_theaters.json?apikey={}".format(YOUR_OWN_KEY))
14    binary = url.content
15    output = json.loads(binary)
16    movies = output['movies']
17    for movie in movies:
18        session.m.append(pulse(movie["title"]))
19    session.m.sort()
20    return TABLE(*[TR(v) for v in session.m]).xml()
21
22 def pulse(movie):
23     text = movie.replace('_', ' ')
24     url = 'http://text-processing.com/api/sentiment/'
25     data = {'text': text}
26     r = requests.post(url, data=data)
27     binary = r.content
28     output = json.loads(binary)
29     label = output["label"]
30     pos = output["probability"]["pos"]
31     neg = output["probability"]["neg"]
32     neutral = output["probability"]["neutral"]
33     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

Figure 21.12: Suggest-a-Movie Example 2

Update Styles

1. Update the child view with some bootstrap styles:

```
1 {{extend 'layout.html'}}
2 <h1>suggest-a-movie</h1>
3 <p>use sentiment to find that perfect movie</p>
4 <br>
5 <form>
6   <input type="button" class="btn btn-primary" value="Get Movies"
7     onclick="ajax('{{URL('grab_movies')}}', [], 'target')">
8 </form>
9 <br>
10 <table class="table table-hover">
11   <thead>
12     <tr>
13       <th>Movie Title</th>
14       <th>Label</th>
15       <th>Positive</th>
16       <th>Negative</th>
17       <th>Neutral</th>
18     </tr>
19   </thead>
20   <tbody id="target"></tbody>
</table>
```

Think about what else you could do? Perhaps you could highlight movies that are positive. Check the web2py [documentation](#) for help.

Deploy to Heroku

Let's deploy this app to Heroku. Simply follow the instructions [here](#).

Check out my app at http://obscure-citadel-4389.herokuapp.com/movie_suggest/default/index.

Cheers!

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

Figure 21.13: Suggest-a-Movie Example 3 - final

Blog App

Let's recreate the blog app we created in Flask. Pay attention, as this will go quickly. Remember the requirements? - The user is presented with the basic user login screen - After logging in, the user can add new posts, read existing posts, or logout - Sessions need to be used to protect against unauthorized users accessing the main page

Once again: activate your virtualenv, start the server, set a password, and enter the Admin Interface. Finally create a new app called "web2blog". (Please Feel free to come up with something a bit more creative. Please.)

Model

The database has one table, *blog_posts*, with two fields - *title* and *post*. Remember how we used an ORM to interact with the database with the Flask framework? Well, web2py uses a similar system called a [Database Abstraction Layer](#) (DAL). To keep things simple, an ORM is a subset of a DAL. Both are used to map database functions to Python objects. Check out the following StackOverflow [article](#) for more info.

Open up *db.py* and append the following code:

```
1 db.define_table('blog_posts',
2     Field('title', notnull=True),
3     Field('post', 'text', notnull=True))
```

We'll go over the main differences between an ORM and a DAL in the next chapter. For now, go ahead and save the file and return to the **Edit** page. As long as there are no errors, web2py created an admin interface used to manage the database, located directly below the "Models" header. Click the button to access the admin interface. From here you can add data to the tables within the database. Go ahead and add a few rows of dummy data, then click on the actual table (db.posts) to view the records you just added.

As soon as the admin interface is accessed, *db.py* is executed and the tables are created. Return to the **Edit** page. A new button should have populated next to the admin interface button called "sql.log". Click the link to view the actual SQL statements used to create the table. Scroll to the bottom.

You should see the following:

```
1 timestamp: 2014-03-03T21:04:08.758346
2 CREATE TABLE blog_posts(
3     id INTEGER PRIMARY KEY AUTOINCREMENT,
4     title CHAR(512) NOT NULL,
```

```

5     post TEXT NOT NULL
6 );
7 success!

```

But what are all those other tables? We'll get to that. One step at a time.

Controller

Next, replace the `index()` function in *default.py* with:

```

1 def index():
2     form=SQLFORM(db.blog_posts)
3     if form.process().accepted:
4         response.flash = "Post accepted - cheers!"
5     elif form.errors:
6         response.flash = "Post not accepted - fix the error(s)."
7     else:
8         response.flash = "Please fill out the form - thank you!"
9     posts = db().select(db.blog_posts.ALL)
10    return dict(posts=posts, form=form)

```

This adds an HTML form so that users can add posts. It also queries the database, pulling all rows of data and returning the results as a dictionary. Again, the values of the dictionary are turned into variables within the views.

View

Edit the view *default/index.html* with the following code:

```

1 {{extend 'layout.html'}}
2 <h1>Add a Post</h1>
3 {{=form}}
4 <h1>Current Posts</h1>
5 <br/>
6 <table>
7     <tr><td><h3>Title</h3></td><td><h3>Post</h3></td></tr>
8     {{for p in posts:}}
9     <tr><td>{{=(A(p.title))}}</td><td>{{=(A(p.post))}}</td></tr>
10    {{pass}}
11 </table>

```

Now, view your app at: <http://127.0.0.1:8000/web2blog/default/>

Wrap Up

All right. So, what are we missing?

1. User login/registration
2. Login_required Decorator to protect *index.html*
3. Session Management

Here's where the real power of web2py comes into play. All of those are auto-implemented in web2py. That's right. Read over it again. Are you starting to like defaults? Remember how long this took to implement into the Flask app?

Simply add the login_required decorator to the index() function in the controller, *default.py*:

```
1 @auth.requires_login()
2 def index():
3     form=SQLFORM(db.blog_posts)
4     if form.process().accepted:
5         response.flash = "Post accepted - cheers!"
6     elif form.errors:
7         response.flash = "Post not accepted - fix the error(s)."
8     else:
9         response.flash = "Please fill out the form - thank you!"
10    posts = db().select(db.blog_posts.ALL)
11    return dict(posts=posts, form=form)
```

Try to access the site again. You should now have to register and login. That's all there is to it! Session Management is already set up as well. I'll explain how this works in the next web2py chapter.

Chapter 22

web2py: py2manager

Introduction

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; it can update the model (POST, PUSH, or PUT requests) and/or views (GET, POST, PUSH, or PUT requests)

Again, a user sends a request to a web server. The server, in turn, passes that request to the controller. Using the established workflow (or logic), the controller then performs an action, such as querying or modifying the database. Once the data is found or updated, the controller then passes the results back to the views, which are rendered into HTML for the end the user to see (response).

Most modern web frameworks utilize the MVC-style architecture, offering similar components. But 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

1. Create a new directory called “py2manager” to house your app.
2. Created and activate a new virtualenv.
3. 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.
4. Create a new app from your terminal:

```
1 $ python web2py.py -S py2manager
```

After this project is created, we are left in the Shell. Go ahead and exit it.

5. Within you terminal, navigate to the “Applications” directory, then into the “py2manager” directory. This directory holds all of your application’s files.

Sublime Text

Instead of using the internal web2py IDE for development, like in the previous examples, we’re going to again use the powerful text editor Sublime Text. This will help us keep our project structure organized.

Load the entire project into Sublime by selecting “Project”, then “Add folder to project.” Navigate to your “/py2manager/applications/py2manager” directory. Select the directory. Click open.

You should now have the entire application structure (files and folders) in Sublime. Take a look around. Open the “Models”, “Views”, and “Controllors” 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.

Let’s start developing!

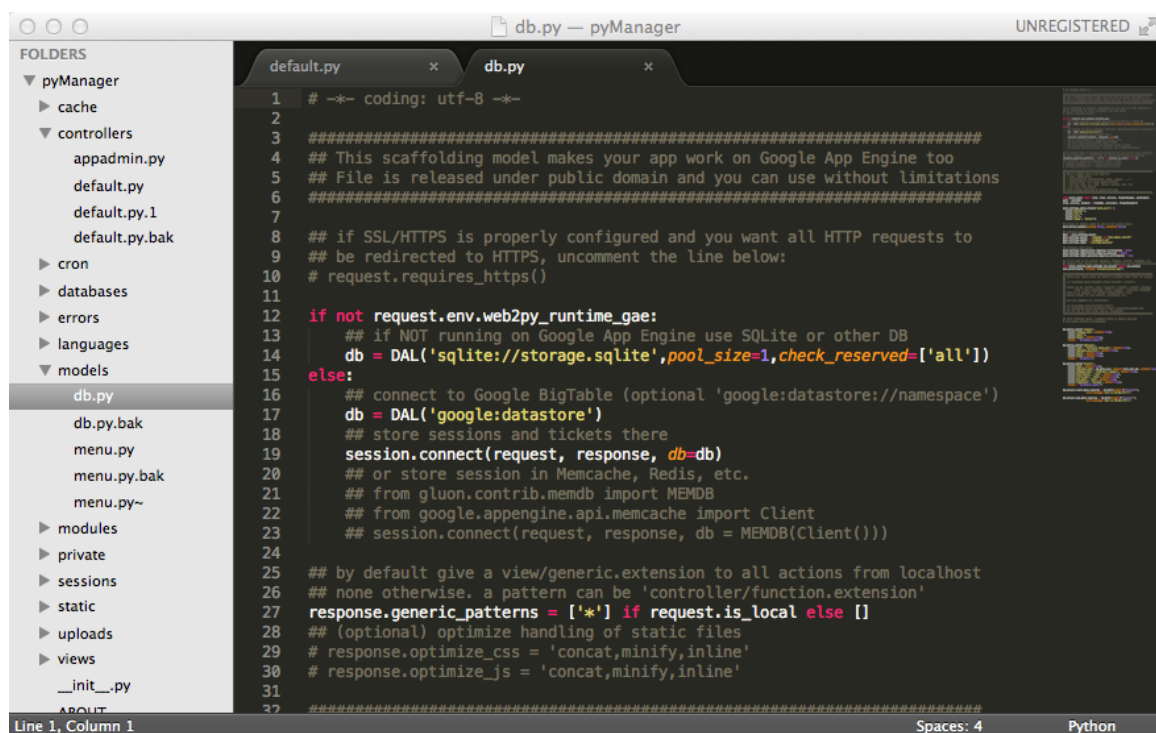


Figure 22.1: Sublime Text with web2py project

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:

```
1 class User(db.Model):
2     __tablename__ = 'users'
3     name = db.Column(db.String, unique=True, nullable=False)
4     email = db.Column(db.String, unique=True, nullable=False)
5     password = db.Column(db.String, nullable=False)
```

DAL:

```
1 db.define_table('users',
2     Field('name', 'string', unique=True, notnull=True),
3     Field('email', 'string', unique=True, notnull=True),
4     Field('password', 'string', 'password', readable=False,
5         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:

1. In your terminal navigate to the project root directory, “/web2py/py2manager”, then run the following command:

```
1 $ python web2py.py --shell=py2manager
```

2. Run the following DAL [commands](#):

```
1 >>> db =
    DAL('sqlite://storage.sqlite',pool_size=1,check_reserved=['all'])
2 >>> db.define_table('special_users', Field('name'), Field('email'))
3 <Table special_users (id,name,email)>
4 >>> db.special_users.insert(id=1, name="Alex", email="hey@alex.com")
5 1L
6 >>> db.special_users.bulk_insert([{'name': 'Alan',
    'email': 'a@a.com'},
7 ...    {'name': 'John', 'email': 'j@j.com'}, {'name': 'Tim',
    'email': 't@t.com'}])
8 [2L, 3L, 4L]
9 >>> db.commit()
10 >>> for row in db().select(db.special_users.ALL):
11 ...     print row.name
12 ...
13 Alex
14 Alan
15 John
16 Tim
17 >>> for row in db().select(db.special_users.ALL):
18 ...     print row
19 ...
20 <Row {'name': 'Alex', 'email': 'hey@alex.com', 'id': 1}>
21 <Row {'name': 'Alan', 'email': 'a@a.com', 'id': 2}>
22 <Row {'name': 'John', 'email': 'j@j.com', 'id': 3}>
23 <Row {'name': 'Tim', 'email': 't@t.com', 'id': 4}>
24 >>> db.special_users.drop()
25 >>> 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 I mentioned in the last chapter, web2py has a default for everything. These are the default values for each table field.

For example:

```
py2manager — bash — 80x27
>>> 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()
(py2manager)Michaels-MacBook-Pro:py2manager michaelherman$
```

Figure 22.2: web2py Shell - adding data to the database

```

1 Field(name, 'string', length=None, default=None,
2       required=False, requires='<default>',
3       ondelete='CASCADE', notnull=False, unique=False,
4       uploadfield=True, widget=None, label=None, comment=None,
5       writable=True, readable=True, update=None, authorize=None,
6       autodelete=False, represent=None, compute=None,
7       uploadfolder=os.path.join(request.folder, 'uploads'),
8       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.

Let's create the model for our application. Navigate into the “applications” directory, and then to the “py2manager” directory.

1. Create a new file to define your database schema called `db_tasks.py` within the “Models” directory. Add the following code:

```

1 db.define_table('company',
2     Field('company_name', notnull=True, unique=True),
3     Field('email'),
4     Field('phone', notnull=True),
5     Field('url'),
6     format = '%(company_name)s')
7
8 db.company.email.requires=IS_EMAIL()
9 db.company.url.requires=IS_EMPTY_OR(IS_URL())
10
11 db.define_table('project',
12     Field('name', notnull=True),
13     Field('employee_name', db.auth_user, default=auth.user_id),
14     Field('company_name', 'reference company', notnull=True),
15     Field('description', 'text', notnull=True),
16     Field('start_date', 'date', notnull=True),
17     Field('due_date', 'date', notnull=True),
18     Field('completed', 'boolean', notnull=True),
19     format = '%(company_name)s')
20
21 db.project.employee_name.readable =
    db.project.employee_name.writable = False

```

We defined a 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, among others. Also, the “employee_name” field in the “project” table references the logged in user. So when a user posts a new project, his/her user information will automatically be added to the database. Save the file.

2. Navigate back to your project root. Fire up web2py in your terminal:

```
1 $ python web2py.py -a 'PUT_YOUR_PASSWORD_HERE' -i 127.0.0.1 -p 8000
```

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`

3. 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:

```
1 timestamp: 2014-03-04T00:44:44.436419
2 CREATE TABLE special_users(
3     id INTEGER PRIMARY KEY AUTOINCREMENT,
4     name CHAR(512),
5     email CHAR(512)
6 );
7 success!
8 DROP TABLE special_users;
9 success!
10 timestamp: 2014-03-04T01:01:27.408170
11 CREATE TABLE auth_user(
12     id INTEGER PRIMARY KEY AUTOINCREMENT,
13     first_name CHAR(128),
14     last_name CHAR(128),
15     email CHAR(512),
16     password CHAR(512),
17     registration_key CHAR(512),
18     reset_password_key CHAR(512),
19     registration_id CHAR(512)
20 );
21 success!
22 timestamp: 2014-03-04T01:01:27.411689
```

```

23 CREATE TABLE auth_group(
24     id INTEGER PRIMARY KEY AUTOINCREMENT,
25     role CHAR(512),
26     description TEXT
27 );
28 success!
29 timestamp: 2014-03-04T01:01:27.413847
30 CREATE TABLE auth_membership(
31     id INTEGER PRIMARY KEY AUTOINCREMENT,
32     user_id INTEGER REFERENCES auth_user (id) ON DELETE CASCADE,
33     group_id INTEGER REFERENCES auth_group (id) ON DELETE CASCADE
34 );
35 success!
36 timestamp: 2014-03-04T01:01:27.416469
37 CREATE TABLE auth_permission(
38     id INTEGER PRIMARY KEY AUTOINCREMENT,
39     group_id INTEGER REFERENCES auth_group (id) ON DELETE CASCADE,
40     name CHAR(512),
41     table_name CHAR(512),
42     record_id INTEGER
43 );
44 success!
45 timestamp: 2014-03-04T01:01:27.419046
46 CREATE TABLE auth_event(
47     id INTEGER PRIMARY KEY AUTOINCREMENT,
48     time_stamp TIMESTAMP,
49     client_ip CHAR(512),
50     user_id INTEGER REFERENCES auth_user (id) ON DELETE CASCADE,
51     origin CHAR(512),
52     description TEXT
53 );
54 success!
55 timestamp: 2014-03-04T01:01:27.421675
56 CREATE TABLE auth_cas(
57     id INTEGER PRIMARY KEY AUTOINCREMENT,
58     user_id INTEGER REFERENCES auth_user (id) ON DELETE CASCADE,
59     created_on TIMESTAMP,
60     service CHAR(512),
61     ticket CHAR(512),
62     renew CHAR(1)

```

```

63 );
64 success!
65 timestamp: 2014-03-04T01:01:27.424647
66 CREATE TABLE company(
67     id INTEGER PRIMARY KEY AUTOINCREMENT,
68     company_name CHAR(512) NOT NULL UNIQUE,
69     email CHAR(512),
70     phone CHAR(512) NOT NULL,
71     url CHAR(512)
72 );
73 success!
74 timestamp: 2014-03-04T01:01:27.427338
75 CREATE TABLE project(
76     id INTEGER PRIMARY KEY AUTOINCREMENT,
77     name CHAR(512) NOT NULL,
78     employee_name INTEGER REFERENCES auth_user (id) ON DELETE
79         CASCADE,
80     company_name INTEGER REFERENCES company (id) ON DELETE CASCADE,
81     description TEXT NOT NULL,
82     start_date DATE NOT NULL,
83     due_date DATE NOT NULL,
84     completed CHAR(1) NOT NULL
85 );
86 success!

```

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.

4. Open <http://localhost:8000/py2manager/default/user/register>, then register yourself as a new user.
5. 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:

```
1 def index():  
2     return dict(message="Hello!")
```

This is just a simple function used to output the string “Hello!” to the screen. You can’t tell from the above info, but the application name is “hello” and the controller used for this function is *default.py*. The function name is “index”.

In this case the generated URL will be:

```
1 http://www.yoursite.com/hello/default/index.html
```

This is also the default URL routing method:

```
1 http://www.yousite.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). If you wanted to remove the controller_name from the URL, for example, add the following code to the newly created file:

```

1 routers = dict(
2     BASE = dict(
3         default_application='py2manager',
4     )
5 )

```

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 setup the logic and URL routing in the py2manager app. Add the following code to *default.py*:

```

1 @auth.requires_login()
2 def index():
3     project_form = SQLFORM(db.project).process()
4     projects = db(db.project).select()
5     users = db(db.auth_user).select()
6     companies = db(db.company).select()
7     return locals()

```

Here we are displaying the data found in the “project”, “auth_user”, and “company” tables, as well as added a form for adding projects.

Most of the functionality is now 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:

1. **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.

For example:

- Add a basic function to the controller:

```
1 def tester():  
2     return locals()
```

- Next, create a new view, “default/tester.html”:

```
1 <html>  
2     <body>  
3         {{numbers = [1, 2, 3]}}  
4         <ul>  
5             {{for n in numbers:}}<li>{{=n}}</li>{{pass}}  
6         </ul>  
7     </body>  
8 </html>
```

- Test it out <http://localhost:8000/tester>.

2. **Template Composition (inheritance):** 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

```
1 {{extend 'layout.html'}}  
2 <html>  
3     <body>  
4         {{numbers = [1, 2, 3]}}
```

```

5     <ul>
6         {{for n in numbers:}}<li>{{=n}}</li>{{pass}}
7     </ul>
8 </body>
9 </html>

```

Test it out again [here](#).

3. Javascript/jQuery libraries: As you have seen, web2py includes a number of Javascript and jQuery libraries, many of which are pre-configured. Refer to the web2py [documentation](#) for more information on Javascript, jQuery, and other components of the views.

Let's build the templates for py2manager.

1. *default/index.html*:

```

1  {{extend 'layout.html'}}
2  <h2>Welcome to py2manager</h2>
3  <br/>
4      {{=(project_form)}}
5  <br/>
6  <h3> All Open Projects </h3>
7  <ul>{{for project in projects:}}
8      <li>
9          {{=(project.name)}}
10         </li>
11         {{pass}}
12 </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*.

2. *default/user.html*:

Open up this file. This template was created automatically to make the development process easier and quicker by providing user authentication in the box. If you go back to the *default.py* file, you can see a description of the main functionalities of the user function:

```

1  """
2  exposes:

```

```

3 http://.../[app]/default/user/login
4 http://.../[app]/default/user/logout
5 http://.../[app]/default/user/register
6 http://.../[app]/default/user/profile
7 http://.../[app]/default/user/retrieve_password
8 http://.../[app]/default/user/change_password
9 use @auth.requires_login()
10     @auth.requires_membership('group name')
11     @auth.requires_permission('read','table name',record_id)
12 to decorate functions that need access control
13 """

```

Read more about authentication [here](#).

3. Layout:

Let's edit the main layout to replace the generic template. Start with *models/menu.py*. Update the following code:

```

1 response.logo = A(B('py',SPAN(2),'manager'),_class="brand")
2 response.title = "py2manager"
3 response.subtitle = T('just another project manager')

```

Then update the application menu:

```

1 response.menu = [(T('Home'), False, URL('default', 'index'), []),
2 (T('Add Project'), False, URL('default', 'add'), []),
3 (T('Add Company'), False, URL('default', 'company'), []),
4 (T('Employees'), False, URL('default', 'employee'), [])]
5
6 DEVELOPMENT_MENU = False

```

Take a look at your changes:

```

1 #####
2 ## Customize your APP title, subtitle and menus here
3 #####
4
5 response.logo = A(B('py',SPAN(2),'manager'),_class="brand")
6 response.title = "py2manager"
7 response.subtitle = T('just another project manager')
8
9 #####

```

```

10 ## this is the main application menu add/remove items as required
11 #####
12
13 response.menu = [(T('Home'), False, URL('default', 'index'), []),
14 (T('Add Project'), False, URL('default', 'add'), []),
15 (T('Add Company'), False, URL('default', 'company'), []),
16 (T('Employees'), False, URL('default', 'employee'), [])]
17
18 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. Again, 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()`:

```
1 auth.settings.extra_fields['auth_user']= [  
2     Field('address'),  
3     Field('city'),  
4     Field('zip'),  
5     Field('image', 'upload')]
```

Save the file. Navigate to <http://localhost:8000/index> and login 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:

Last name:

E-mail:

Address:

City:

Zip:

No file chosen [\[file\]](#) ☐ delete


Image: 

Figure 22.3: py2manager Profile page

Add Projects

To clean up the homepage, let's move the form to add new projects to a separate page.

1. Open your *default.py* file, and add a new function:

```
1 @auth.requires_login()
2 def add():
3     project_form = SQLFORM(db.project).process()
4     return dict(project_form=project_form)
```

2. Then update the *index()* function:

```
1 @auth.requires_login()
2 def index():
3     projects = db(db.project).select()
4     users = db(db.auth_user).select()
5     companies = db(db.company).select()
6     return locals()
```

3. Add a new template in the default directory called *add.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Add a new project:</h2>
3 <br/>
4 {{=project_form.custom.begin}}
5 <strong>Project
6     name</strong><br/>{{=project_form.custom.widget.name}}<br/>
7 <strong>Company
8     name</strong><br/>{{=project_form.custom.widget.company_name}}<br/>
9 <strong>Description</strong><br/>
10 {{=project_form.custom.widget.description}}<br/>
11 <strong>Start
12     Date</strong><br/>{{=project_form.custom.widget.start_date}}<br/>
13 <strong>Due
14     Date</strong><br/>{{=project_form.custom.widget.due_date}}<br/>
15 {{=project_form.custom.submit}}
16 {{=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]`.

4. Remove the form from *index.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Welcome to py2manager</h2>
3 <br>
4 <h3> All Open Projects </h3>
5 <ul>{{for project in projects:}}
6     <li>
7         {{=(project.name)}}
8     </li>
9     {{pass}}
10 </ul>
```

Add Companies

We need to add a form for adding new companies, which follows almost a nearly identical pattern as adding a form for projects. Try working on it on your own before looking at the code.

1. *default.py*:

```
1 @auth.requires_login()
2 def company():
3     company_form = SQLFORM(db.company).process()
4     return dict(company_form=company_form)
```

2. Add a new template in the default directory called *company.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Add a new company:</h2>
3 <br/>
4 {{=company_form.custom.begin}}
5 <strong>Company
6     Name</strong><br/>{{=company_form.custom.widget.company_name}}<br/>
7 <strong>Email</strong><br/>{{=company_form.custom.widget.email}}<br/>
8 <strong>Phone</strong><br/>{{=company_form.custom.widget.phone}}<br/>
9 <strong>URL</strong><br/>{{=company_form.custom.widget.url}}<br/>
10 {{=company_form.custom.submit}}
11 {{=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.

1. Update the `index()` function in `default.py`:

```
1 @auth.requires_login()
2 def index():
3     response.flash = T('Welcome!')
4     grid = SQLFORM.grid(db.project)
5     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.

2. Update the `index.html` view:

```
1 {{extend 'layout.html'}}
2 <h2>All projects:</h2>
3 <br/>
4 {{=grid}}
```

3. Navigate to <http://127.0.0.1:8000/> 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.
4. Append the following code to the bottom of `db_tasks.py`:

```
1 db.project.start_date.requires = IS_DATE(format=T('%m-%d-%Y'),
2     error_message='Must be MM-DD-YYYY!')
3
4 db.project.due_date.requires = IS_DATE(format=T('%m-%d-%Y'),
5     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/add>. Use the built-in AJAX calendar. *Oops*. That's still inputting dates the old way. Let's fix that.

5. Within the “views” folder, open *web2py_ajax.html* and make the following changes:

Change:

```
1 var w2p_ajax_date_format = "{{=T('%Y-%m-%d')}}";  
2 var w2p_ajax_datetime_format = "{{=T('%Y-%m-%d %H:%M:%S')}}";
```

To:

```
1 var w2p_ajax_date_format = "{{=T('%m-%d-%Y')}}";  
2 var w2p_ajax_datetime_format = "{{=T('%m-%d-%Y %H:%M:%S')}}";
```

6. Now let's update the grid in the `index()` function within the controller:

```
1 grid = SQLFORM.grid(db.project, create=False,  
2     fields=[db.project.name, db.project.employee_name,  
3     db.project.company_name, db.project.start_date,  
4     db.project.due_date, db.project.completed],  
5     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.

1. *default.py*:

```
1 @auth.requires_login()
2 def company():
3     company_form = SQLFORM(db.company).process()
4     grid = SQLFORM.grid(db.company, create=False, deletable=False,
5         editable=False,
6         maxtextlength=50, orderby=db.company.company_name)
7     return locals()
```

2. *company.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Add a new company:</h2>
3 <br/>
4 {{=company_form.custom.begin}}
5 <strong>Company
6     Name</strong><br/>{{=company_form.custom.widget.company_name}}<br/>
7 <strong>Email</strong><br/>{{=company_form.custom.widget.email}}<br/>
8 <strong>Phone</strong><br/>{{=company_form.custom.widget.phone}}<br/>
9 {{=company_form.custom.submit}}
10 {{=company_form.custom.end}}
11 <br/>
12 <h2>All companies:</h2>
13 <br/>
14 {{=grid}}
```

Next, let's create the employee view:

1. *default.py*:

```
1 @auth.requires_login()
2 def employee():
3     employee_form = SQLFORM(db.auth_user).process()
```

```
4     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)
5     return locals()
```

2. *employee.html*:

```
1     {{extend 'layout.html'}}
2     <h2>All employees:</h2>
3     <br/>
4     {{=grid}}
```

Test both of the new views in the browser.

Notes

Next, let's add the ability to add notes to each project.

1. Add a new table to the database in *db_task.py*:

```
1 db.define_table('note',
2     Field('post_id', 'reference project', writable=False),
3     Field('post', 'text', notnull=True),
4     Field('created_on', 'datetime', default=request.now,
5         writable=False),
6     Field('created_by', db.auth_user, default=auth.user_id))
7
8 db.note.post_id.readable = db.note.post_id.writable = False
9 db.note.created_on.readable = db.note.created_on.writable = False
10 db.note.created_on.requires = IS_DATE(format=T('%m-%d-%Y'),
11     error_message='Must be MM-DD-YYYY!')
12 db.note.created_by.readable = db.note.created_by.writable = False
```

2. Update the `index()` function and add a `note()` function in the controller:

```
1 @auth.requires_login()
2 def index():
3     response.flash = T('Welcome!')
4     notes = [lambda project:
5         A('Notes', _href=URL("default", "note", args=[project.id]))]
6     grid = SQLFORM.grid(db.project, create=False, links=notes,
7         fields=[db.project.name, db.project.employee_name,
8             db.project.company_name, db.project.start_date,
9             db.project.due_date, db.project.completed],
10         deletable=False, maxtextlength=50)
11     return locals()
12
13 @auth.requires_login()
14 def note():
15     project = db.project(request.args(0))
16     db.note.post_id.default = project.id
17     form = crud.create(db.note) if auth.user else "Login to Post to
18         the Project"
19     allnotes = db(db.note.post_id==project.id).select()
20     return locals()
```

3. Take a look. Add some notes. Now let's add a new view called *default/note.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Project Notes</h2>
3 <br/>
4 <h4>Current Notes</h4>
5 {{for n in allnotes:}}
6     <ul>
7         <li>{{=db.auth_user[n.created_by].first_name}} on
8             {{=n.created_on.strftime("%m/%d/%Y")}}
9             - {{=n.post}}</li>
10    </ul>
11 {{pass}}
12 <h4>Add a note</h4>
13 {{=form}}<br>
```

4. Finally, let's update the `index()` function to add a button for the Notes link:

```
1 @auth.requires_login()
2 def index():
3     response.flash = T('Welcome!')
4     notes = [lambda project: A('Notes', _class="btn",
5                               _href=URL("default", "note", args=[project.id]))]
6     grid = SQLFORM.grid(db.project, create=False, links=notes,
7                         fields=[db.project.name, db.project.employee_name,
8                               db.project.company_name,
9                               db.project.start_date, db.project.due_date,
10                               db.project.completed], deletable=False, maxtextlength=50)
11     return locals()
```

We just added the `btn` class to the `notes` variable.

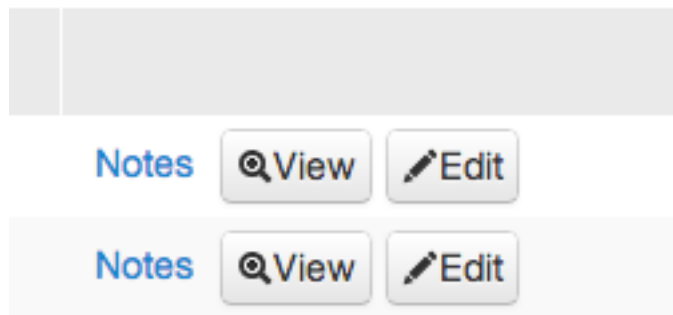


Figure 22.4: py2manager - grid links (just added Notes button)

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, so add the following code to the `routes.py` file:

```
1 routes_onerror = [  
2     ('*/*', '/py2manager/static/error.html')  
3 ]
```

Then add the `error.html` file to the “static” directory:

```
1 <h2>This is an error. We are working on fixing it.</h2>
```

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.

Final Word

What's this app missing? Would you like to see any additional features? If so, please post your feedback on the Real Python [forum](#). Thanks!

Chapter 23

Interlude: Web Scraping and Crawling

NOTE: This chapter utilizes Python 3.5; however, all scripts are compatible with both Python 2 and 3.

Since data is unfortunately not always accessible through a manageable format via web APIs (i.e, XML or JSON), 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 act of actual 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 span of time to a website's server, which may use up bandwidth, slowing down the website for other users and potentially overloading the server; and
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:

Access and Interference

eBay's sites contain robot exclusion headers. Information on our sites is subject to constant updates and changes. Much of the information on the sites is also proprietary or is licensed to eBay by our users or third parties. You agree that you will not use any robot, spider, scraper, or other automated means to access our sites for any purpose without our express handwritten permission.

Additionally, you agree that you will not:

- take any action that imposes or may impose (to be determined in our sole discretion) an unreasonable or disproportionately large load on our infrastructure;
- copy, reproduce, reverse engineer, modify, create derivative works from, distribute, or publicly display any content (except for your information) from our sites, services, applications, or tools without the prior express written permission of eBay and the appropriate third party, as applicable;
- interfere or attempt to interfere with the proper working of our sites, services, applications, or tools, or any activities conducted on or with our sites, services, applications, or tools; or
- bypass our robot exclusion headers or other measures we may use to prevent or restrict access to our sites.

Figure 23.1: Ebay Terms of Service

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.

3. 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
```

Figure 23.2: HackerNews' robots.txt

- 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 crawling 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 five to ten 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==1.0.3`

NOTE If you are using Windows 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 - 1.0.3. Good luck!

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”, and then start a new Scrapy project:

```
1 $ scrapy startproject hackernews
2 2015-11-21 09:28:53 [scrapy] INFO: Scrapy 1.0.1 started (bot:
   scrapybot)
3 2015-11-21 09:28:53 [scrapy] INFO: Optional features available:
   ssl, http11
4 2015-11-21 09:28:53 [scrapy] INFO: Overridden settings: {}
5 New Scrapy project 'hackernews' created in:
6   /Users/michaelherman/repos/realpython/book2-exercises/scraping/hackernews
7
8 You can start your first spider with:
9   cd hackernews
10  scrapy genspider example example.com
```

This will create a “hackernews” directory with the following contents:

```
1
2 hackernews
3   __init__.py
4   items.py
5   pipelines.py
6   settings.py
7   spiders
8   __init__.py
9 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:

```
1 $ cd hackernews
2 $ 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:

```

1 import scrapy
2
3
4 class HackernewsItem(scrapy.Item):
5     title = scrapy.Field()
6     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"):

```

1 from scrapy import Spider
2 from scrapy.selector import Selector
3
4 from hackernews.items import HackernewsItem
5
6
7 class BasicSpider(Spider):
8     # name the spider
9     name = "basic"
10
11     # allowed domains to scrape
12     allowed_domains = ["news.ycombinator.com"]
13
14     # urls the spider begins to crawl from
15     start_urls = (
16         'https://news.ycombinator.com/',
17     )
18
19     # parses and returns the scraped data
20     def parse(self, response):
21         titles =
22             Selector(response).xpath('//tr[@class="athing"]/td[3]')
23
24         for title in titles:
25             item = HackernewsItem()
26             item['title'] = title.xpath("a[@href]/text()").extract()
27             item['url'] = title.xpath("a/@href").extract()
28             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`.

All right. So what's going on here?

Essentially, you 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?

1. Open the start url in FireFox or Chrome: <https://news.ycombinator.com/>
2. Right click on the first article link and select "Inspect Element"
3. In the Firebug or Chrome Developer Tools console, you can see the HTML that's used to display the first link:

```
1 <tr class="athing">
2   <td align="right" valign="top" class="title">
3     <span class="rank">1.</span>
4   </td>
5   <td valign="top" class="votelinks">
6     <center>
7       <a id="up_10606355" onclick="return vote(this)"
8         href="vote?for=10606355&dir=up&auth=2febc45204aa28f15f87c085df8fcfe96a99d85d&goto=news"><div
9         class="votearrow" title="upvote"></div></a>
10    </center>
11  </td>
12  <td class="title">
13    <span class="deadmark"></span>
14    <a href="https://mikeash.com/pyblog/friday-qa-2015-11-20-covariance-and-contravariance.html">Covariance
    and Contravariance</a>
```

```

15     <span class="sitebit comhead"> (<a
      href="from?site=mikeash.com"><span
      class="sitestr">mikeash.com</span></a></span>
16   </td>
17 </tr>

```

4. You can see that everything we need, text and url, is located between the `<td class="title">` `</td>` tag:

```

1   <td class="title">
2   <span class="deadmark"></span>
3   <a href="https://mikeash.com/pyblog/
4   friday-qa-2015-11-20-covariance-and-contravariance.html">Covariance
      and Contravariance</a>
5   <span class="sitebit comhead"> (<a
      href="from?site=mikeash.com"><span
      class="sitestr">mikeash.com</span></a></span>
6   </td>

```

And if you look at the rest of the document, all other postings fall within the same tag.

5. Thus, we have our main XPath:

```

1 titles = Selector(response).xpath('//tr[@class="athing"]/td[3]')

```

6. Now, we just need to establish the XPath for the title and url. Take a look at the HTML again:

```

1 <a href="https://mikeash.com/pyblog/
2 friday-qa-2015-11-20-covariance-and-contravariance.html">Covariance
      and Contravariance</a>

```

7. 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 is `scrapy shell <url>`. Open your terminal and type `scrapy shell http://news.ycombinator.com`. Assuming there are no errors in the URL, you can now test your XPath expressions.

1. Start by using Firebug or 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.
2. Type `sel.xpath('//tr[@class="athing"]/td[3]').extract()` in the Shell and press enter.
3. It's hard to tell, but the URL and title are both part of the results. We're on the right path.
4. Add the `a` to the test:

```
1 sel.xpath('//tr[@class="athing"]/td[3]/a').extract()[0]
```

NOTE: By adding `[0]` to the end, we are just returning the first result.

5. Now you can see that just the title and URL are part of the results. Now, just extract the text and then the href:

```
1 sel.xpath('//tr[@class="athing"]/td[3]/a/text()').extract()[0]
```

and

```
1 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 data that you want to scrape.

Try some more XPath expressions:

- The “library” link at the bottom of the page:

```
1 sel.xpath('//span[@class="yclinks"]/a[3]/@href').extract()
```

- The comment links and URLs:

```
1 sel.xpath('//td[@class="subtext"]/a/@href').extract()[0]
```

and

```
1 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 be scraping a listing of new movies from Wikipedia: http://en.wikipedia.org/wiki/Category:2014_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, right?

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.

1. Start a new Scrapy project in the “scraping” directory:

```
1 $ scrapy startproject wikipedia
```

2. Create the *items.py* file:

```
1 import scrapy
2
3
4 class WikipediaItem(scrapy.Item):
5     title = scrapy.Field()
6     url = scrapy.Field()
```

3. Setup your crawler. Again, you can setup a skeleton crawler using the following command:

```
1 $ cd wikipedia
2 $ scrapy genspider wiki en.wikipedia.org
```

4. Finish coding the scraper:


```

1 from scrapy import Spider
2 from scrapy.selector import Selector
3
4 from wikipedia.items import WikipediaItem
5
6
7 class BasicSpider(Spider):
8     # name the spider
9     name = "wiki"
10
11     # allowed domains to scrape
12     allowed_domains = ["en.wikipedia.org"]
13
14     # urls the spider begins to crawl from
15     start_urls = (
16         "http://en.wikipedia.org/wiki/Category:2014_films",
17     )
18
19     # parses and returns the scraped data
20     def parse(self, response):
21
22         titles =
23             Selector(response).xpath('//div[@id="mw-pages"]//li')
24
25         for title in titles:
26             item = WikipediaItem()
27             url = title.select("a/@href").extract()
28             item["title"] = title.select("a/text()").extract()
29             item["url"] = url[0]
30             yield item

```

Save this to your “spiders” directory as *wiki.py*.

Did you notice the XPath?

```
1 hxs.select('//div[@id="mw-pages"]//li')
```

This is equivalent to:

```
1 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:

```
1 $ 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:

```
1 for title in titles:
2     item = WikipediaItem()
3     url = title.select("a/@href").extract()
4     item["title"] = title.select("a/text()").extract()
5     item["url"] = urljoin("http://en.wikipedia.org", url[0])
6     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:

```
1 for title in titles:
2     item = WikipediaItem()
3     url = title.select("a/@href").extract()
4     if url:
5         item["title"] = title.select("a/text()").extract()
6         item["url"] = urljoin("http://en.wikipedia.org", url[0])
7         yield item
```

Your script should look like this:

```
1 from scrapy import Spider
2 from scrapy.selector import Selector
3 from urlparse import urljoin
4
5 from wikipedia.items import WikipediaItem
6
7
8 class BasicSpider(Spider):
9     # name the spider
10     name = "wiki"
11
```

```

12     # allowed domains to scrape
13     allowed_domains = ["en.wikipedia.org"]
14
15     # urls the spider begins to crawl from
16     start_urls = (
17         "http://en.wikipedia.org/wiki/Category:2014_films",
18     )
19
20     # parses and returns the scraped data
21     def parse(self, response):
22
23         titles =
24             Selector(response).xpath('//div[@id="mw-pages"]//li')
25
26         for title in titles:
27             item = WikipediaItem()
28             url = title.select("a/@href").extract()
29             if url:
30                 item["title"] = title.select("a/text()").extract()
31                 item["url"] = urljoin("http://en.wikipedia.org",
                                         url[0])
32                 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 be scraping a listing of publicly available datasets from Socrata: <https://opendata.socrata.com/>.

Create the project:

```
1 $ 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.

1. *items.py*:

```
1 import scrapy
2
3
4 class SocrataItem(scrapy.Item):
5     text = scrapy.Field()
6     url = scrapy.Field()
7     views = scrapy.Field()
```

2. Create the generic spider:

```
1 $ cd socrata
2 $ scrapy genspider opendata opendata.socrata.com
```

3. *opendata.py*:

```
1 from scrapy import Spider
2 from scrapy.selector import Selector
3
4 from socrata.items import SocrataItem
5
6
7 class OpendataSpider(Spider):
8     name = "opendata"
9     allowed_domains = ["opendata.socrata.com"]
10    start_urls = (
```

```

11         'https://opendata.socrata.com/',
12     )
13
14     def parse(self, response):
15         titles =
16             Selector(response).xpath('//tr[@itemscope="itemscope"]')
17         for title in titles:
18             item = SocrataItem()
19             item["text"] =
20                 title.select("td[2]/div/span/text()").extract()
21             item["url"] =
22                 title.select("td[2]/div/a/@href").extract()
23             item["views"] =
24                 title.select("td[3]/span/text()").extract()
25             yield item

```

4. Release the spider:

```

1 $ scrapy crawl opendata -o socrata.json

```

5. 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.

1. First, there's no change to *items.py*. We will be scraping the same data on each page.
2. Make a copy of *opendata.py*. Save it as *opendata_crawl.py*.
3. Update the imports:

```

1 from scrapy.linkextractors import LinkExtractor
2 from scrapy.spiders import CrawlSpider, Rule
3
4 from socrata.items import SocrataItem

```

4. Update the name: `name = "opendatacrawl"`

5. Add the rules just below the `start_urls`:

```
1 rules = [  
2     Rule(LinkExtractor(allow='browse\?utf8=%E2%9C%93&page=\d*'),  
3         callback='parse_item', follow=True)  
4 ]
```

6. What else do you have to update? First, the class must inherit from `CrawlSpider`, not `Spider`. Anything else?

```
1 from scrapy.linkextractors import LinkExtractor  
2 from scrapy.spiders import CrawlSpider, Rule  
3  
4 from socrata.items import SocrataItem  
5  
6  
7 class OpendataSpider(CrawlSpider):  
8     name = "opendatacrawl"  
9     allowed_domains = ["opendata.socrata.com"]  
10    start_urls = (  
11        'https://opendata.socrata.com/',  
12    )  
13    rules = [  
14        Rule(LinkExtractor(allow='browse\?utf8=%E2%9C%93&page=\d*'),  
15            callback='parse_item', follow=True)  
16    ]  
17  
18    def parse(self, response):  
19        titles = response.xpath('//tr[@itemscope="itemscope"]')  
20        for title in titles:  
21            item = SocrataItem()  
22            item["text"] =  
23                title.select("td[2]/div/span/text()").extract()  
24            item["url"] =  
25                title.select("td[2]/div/a/@href").extract()  
26            item["views"] =  
27                title.select("td[3]/span/text()").extract()  
28            yield item
```

As you can see, the only new part of the code, besides the imports, are the rules, which define the crawling portion of the spider:

```
1 rules = [  
2     Rule(LinkExtractor(allow='browse\?utf8=%E2%9C%93&page=\d*'),  
3         callback='parse_item', follow=True)  
4 ]
```

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:

```
1 browse\?utf8=%E2%9C%93&page=\d*
```

Make sense?

All right. Remember how I 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. Or 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:

```
1 DOWNLOAD_DELAY = 10
```

Item Pipeline

Finally, instead of dumping the data into a JSON file, let's feed it to a database.

1. Create the database within the *first* "socrata" directory from your shell:

```
1 $ python
2 >>> import sqlite3
3 >>> conn = sqlite3.connect("project.db")
4 >>> cursor = conn.cursor()
5 >>> cursor.execute("CREATE TABLE data(text TEXT, url TEXT, views
6 >>> TEXT)")
7 <sqlite3.Cursor object at 0x1029db730>
```

2. Update the *pipelines.py* file:

```
1 import sqlite3
2
3
4 class SocrataPipeline(object):
5     def __init__(self):
6         self.conn = sqlite3.connect('project.db')
7         self.cur = self.conn.cursor()
8
9     def process_item(self, item, spider):
10        self.cur.execute(
11            "insert into data (text, url, views) values(?,?,?)",
12            (item['text'][0], item['url'][0], item['views'][0])
13        )
14        self.conn.commit()
15        return item
```


3. Add the pipeline to the *settings.py* file:

```
1 ITEM_PIPELINES = {'socrata.pipelines.SocrataPipeline': 300}
```

4. Test this out with the BaseSpider first:

```
1 $ scrapy crawl opendata
```

Look good? Go ahead and delete the data using the SQLite Browser. Save the database.

Ready? Fire away:

```
1 $ scrapy crawl opendata_crawl
```

This will take a while. In the meantime, read about using Firebug with Scrapy from the official Scrapy [documentation](#). Still running? Take a break. Stretch. Do a little dance.

Once complete, open the database with the SQLite Browser. You should have about ~20,000 rows of data. Make sure to hold onto this database we'll be using it later.

For now, go ahead and move on with the scraper running.

Homework

- Use your knowledge of BeautifulSoup, which, again, was taught in the first [Real Python](#) course, as well the 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 main course or the BeautifulSoup [documentation](#) for assistance.

Use the following command to install BeautifulSoup: `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 into 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.

```
1 # Download stock quotes in CSV
2
3
4 import requests
5 import time
6
7 i = 0
8
9 # obtain quote once every 3 seconds for the next 6 seconds
10 while (i < 2):
11
12     # define the base url
13     base_url = 'http://download.finance.yahoo.com/d/quotes.csv'
14
15     # retrieve data from web server
16     data = requests.get(
17         base_url,
18         params={'s': 'GOOG', 'f': 'sl1d1t1c1ohgv', 'e': '.csv'})
19
20     # write the data to csv
21     with open('stocks.csv', 'wb') as code:
22         code.write(data.content)
23
24     i += 1
25
26     # pause for 3 seconds
27     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 I got the parameters: `params={'s': 'GOOG', 'f': 'sl1d1t1c1ohgv', 'e': '.csv'}}`

Open <http://download.finance.yahoo.com/> in your browser and search for the stock quote for Google: GOOG. Then copy the url for downloading the spreadsheet:

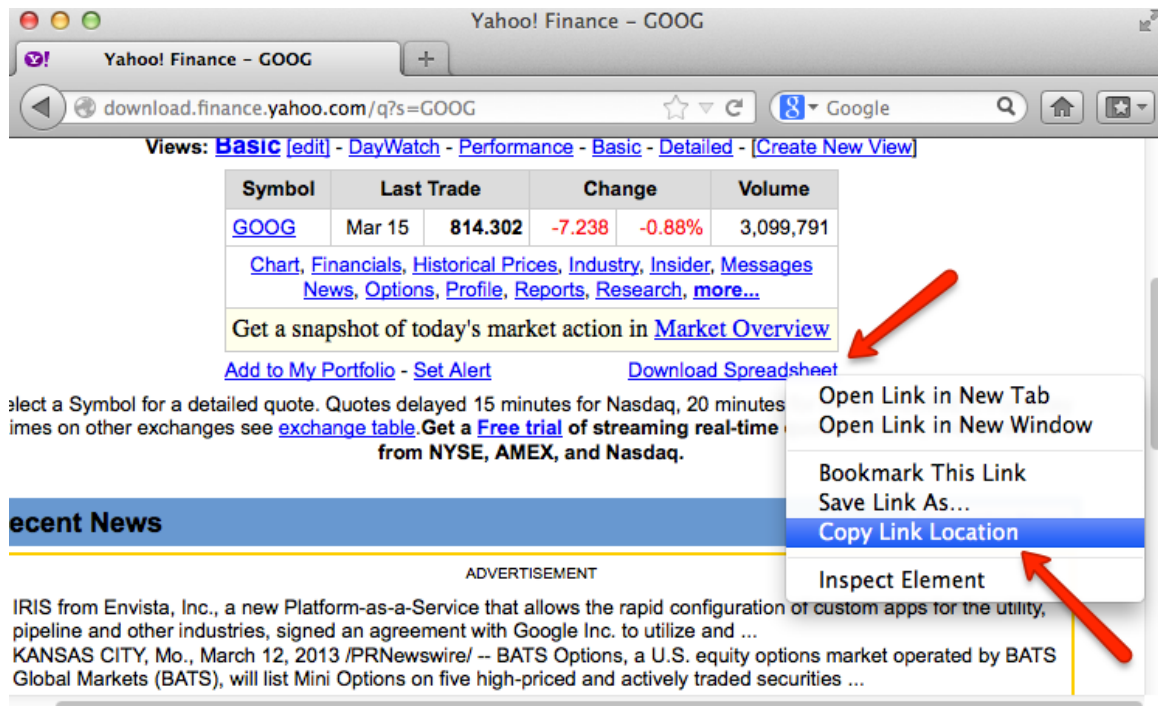


Figure 23.3: Yahoo Finance - grab url for the CSV

<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.

```
1 # Download stock quotes in CSV
2
3
4 import requests
```

```

5 import time
6
7 i = 0
8
9 # defint the stocks to download
10 stock_list = ['GOOG', 'YHOO', 'MSF']
11
12 while (i < 1):
13
14     # define the base url
15     base_url = 'http://download.finance.yahoo.com/d/quotes.csv'
16
17     # retrieve data from web server
18     for stock in stock_list:
19         data = requests.get(
20             base_url,
21             params={'s': stock, 'f': 'sl1d1t1c1ohgv', 'e': '.csv'}
22         )
23
24     # output the data to the screen
25     print(data.content)
26
27     i += 1
28
29     # pause for 3 seconds
30     time.sleep(3)

```

Chapter 24

web2py: REST Redux

Introduction

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. In short, 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 REST, 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 on the website. 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. Perhaps you are scraping other websites (legally) that also have data sources and you're creating an aggregator service.

Whatever the reason, let's look at how to quickly setup a RESTful web service via web2py by to expose the data we pulled.

Remember:

- Each resource or endpoint should be identified by a separate URL.

- There are four HTTP methods used for interacting with Databases (CRUD):
- read (GET)
- create (POST)
- update (PUT)
- delete (DELETE).

Let's start with a basic example before using the scraped data from Socrata.

Basic REST

Create a new folder called “web2py-rest”. Then install and activate a virtualenv. Download the source files from web2py. Unzip the file. Add the contents directly to “web2py-rest”. Fire up the server. Then create a new application called “basic_rest”.

To set up a RESTful API follow these steps:

1. Create a new database table in *db.py*:

```
1 db.define_table('fam',Field('role'),Field('name'))
```

2. Within the web2py admin, enter some dummy data into the newly created table.

3. Add the RESTful functions to the [controller](#):

```
1 @request.restful()
2 def api():
3     response.view = 'generic.'+request.extension
4     def GET(*args,**vars):
5         patterns = 'auto'
6         parser = db.parse_as_rest(patterns,args,vars)
7         if parser.status == 200:
8             return dict(content=parser.response)
9         else:
10            raise HTTP(parser.status,parser.error)
11     def POST(table_name,**vars):
12         return db[table_name].validate_and_insert(**vars)
13     def PUT(table_name,record_id,**vars):
14         return db(db[table_name]._id==record_id).update(**vars)
15     def DELETE(table_name,record_id):
16         return db(db[table_name]._id==record_id).delete()
17     return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)
```

These functions expose any field in our database to the outside world. If you want to limit the resources exposed, you’ll need to define various [patterns](#).

For example:

```
1 def GET(*args,**vars):
2     patterns = [
3         "/test[fam]",
4         "/test/{fam.name.startswith}",
```

```

5     "/test/{fam.name}/:field",
6 ]
7 parser = db.parse_as_rest(patterns,args,vars)
8 if parser.status == 200:
9     return dict(content=parser.response)
10 else:
11     raise HTTP(parser.status,parser.error)

```

So, 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.

4. Test out the following GET requests in your browser:

- URI: `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" }] }`

Explanation: GET all data

- URI: `http://127.0.0.1:8000/basic_rest/default/api/fam/id/1.json`

```
1 { "content": [ { "role": "Father", "id": 1, "name": "Tom" } ] }
```

Explanation: GET data where "id" == 1

NOTE: These results will vary depending upon the sample data that you added.

5. Test out the following requests in the Shell and look at the results in the database:

```

1 >>> import requests
2 >>>
3 >>> payload = { "name" : "john", "role" : "brother" }

```



```

4 >>> r =
    requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json",
        data=payload)
5 >>> r
6 <Response [200]>
7 >>>
8 >>> r =
    requests.delete("http://127.0.0.1:8000/basic_rest/default/api/fam/2.json")
9 >>> r
10 <Response [200]>
11 >>>
12 >>> payload = {"name" : "Jeffrey"}
13 >>> r =
    requests.put("http://127.0.0.1:8000/basic_rest/default/api/fam/3.json",
        data=payload)
14 >>> r
15 <Response [200]>

```

6. Now 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.
7. 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:

```

1 auth.settings.allow_basic_login = True
2
3 @auth.requires_login()
4 @request.restful()
5 def api():
6     response.view = 'generic.'+request.extension
7     def GET(*args,**vars):
8         patterns = 'auto'
9         parser = db.parse_as_rest(patterns,args,vars)
10        if parser.status == 200:
11            return dict(content=parser.response)
12        else:
13            raise HTTP(parser.status,parser.error)
14    def POST(table_name,**vars):
15        return db[table_name].validate_and_insert(**vars)
16    def PUT(table_name,record_id,**vars):

```

```

17     return db(db[table_name]._id==record_id).update(**vars)
18 def DELETE(table_name,record_id):
19     return db(db[table_name]._id==record_id).delete()
20 return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)

```

8. Now you need to be authenticated to make any requests:

```

1 >>> import requests
2 >>> from requests.auth import HTTPBasicAuth
3 >>> payload = {"name" : "Katie", "role" : "Cousin"}
4 >>> auth = HTTPBasicAuth("Michael", "reallyWRONG")
5 >>> r =
    requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json",
    data=payload, auth=auth)
6 >>> r
7 <Response [403]>
8 >>>
9 >>> auth = HTTPBasicAuth("michael@realpython.com", "admin")
10 >>> r =
    requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json",
    data=payload, auth=auth)
11 >>> r
12 <Response [200]>

```

9. Test this out some more.

Homework

- Please watch [this](#) short video on REST.

Advanced REST

All right. 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

1. Create a new app called "socrata" within "web2py-rest"
2. Register a new user: <http://127.0.0.1:8000/socrata/default/user/register>
3. Create a new table with the following schema:

```
1 db.define_table('socrata',Field('name'),Field('url'),Field('views'))
```

4. 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`
- You need to rename the "text" field since it's technically a restricted name. Change this to "name". Then use the Find and Replace feature in Sublime to remove the word "views" from each data point. The word "views" makes it impossible to filter and sort the data. When we scraped the data, we should have only grabbed the view count, not the word "views". Mistakes are how you learn, though. Make sure when you do your find, grab the blank space in front of the word as well - e.g, " views".
- 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:

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

1. First, 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/resources,
- JSON should be the format of choice, and
- Use only lowercase characters.

2. Add the following code to *default.py*:

```
1 @request.restful()
2 def api():
3     response.view = 'generic.json'+request.extension
4     def GET(*args,**vars):
5         patterns = 'auto'
6         parser = db.parse_as_rest(patterns,args,vars)
7         if parser.status == 200:
8             return dict(content=parser.response)
9         else:
10            raise HTTP(parser.status,parser.error)
11     def POST(table_name,**vars):
12         return db[table_name].validate_and_insert(**vars)
13     def PUT(table_name,record_id,**vars):
14         return db(db[table_name]._id==record_id).update(**vars)
15     def DELETE(table_name,record_id):
16         return db(db[table_name]._id==record_id).delete()
17     return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)
```

Test

GET

Navigate to the following URL to see the resources/end points that are available via GET:

<http://127.0.0.1:8000/socrata/default/api/patterns.json>

Output:

```

1 {
2   content: [
3     "/socrata[socrata]",
4     "/socrata/id/{socrata.id}",
5     "/socrata/id/{socrata.id}/:field"
6   ]
7 }

```

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].json)
- [http://127.0.0.1:8000/socrata/default/api/socrata/id/\[id\]/\[field_name\].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 the requests library to start:

```

1 >>> import requests

```

1. [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].json)

```

1 >>> r = requests.get("http://127.0.0.1:8000/socrata/
2   default/api/socrata/id/1.json")
3 >>> r
4 <Response [200]>
5 >>> r.content
6 '{"content": [{"name": "Drug and Alcohol Treatments Florida",
7   "views": "6", "url": "https://opendata.socrata.com/
8   dataset/Drug-and-Alcohol-Treatments-Florida/uzmv-9jrm", "id":
   100}]]\n'

```

2. [http://127.0.0.1:8000/socrata/default/api/socrata/id/\[id\]/\[field_name\].json](http://127.0.0.1:8000/socrata/default/api/socrata/id/[id]/[field_name].json)

```

1 >>> r = requests.get("http://127.0.0.1:8000/socrata/
2   default/api/socrata/id/100/name.json")
3 >>> r
4 <Response [200]>
5 >>> r.content

```

```

6 '{"content": [{"name": "Drug and Alcohol Treatments Florida"}]}\n'
7 >>> r = requests.get("http://127.0.0.1:8000/
8     socrata/default/api/socrata/id/100/views.json")
9 >>> r
10 <Response [200]>
11 >>> r.content
12 '{"content": [{"views": "6"}]}\n'

```

POST

http://127.0.0.1:8000/socrata/default/api/socrata.json

```

1 >>> payload = {'name': 'new
2     database', 'url': 'http://new.com', 'views': '22'}
3 >>> r = requests.post("http://127.0.0.1:8000/
4     socrata/default/api/socrata.json", payload)
5 >>> r
<Response [200]>

```

PUT

http://127.0.0.1:8000/socrata/default/api/socrata/[id].json

```

1 >>> payload = {'name': 'new database'}
2 >>> r = requests.put("http://127.0.0.1:8000/
3     socrata/default/api/socrata/3.json", payload)
4 >>> r
5 <Response [200]>
6 >>> r = requests.get("http://127.0.0.1:8000/
7     socrata/default/api/socrata/id/3/name.json")
8 >>> r
9 <Response [200]>
10 >>> r.content
11 '{"content": [{"name": "new database"}]}\n'

```

DELETE

http://127.0.0.1:8000/socrata/default/api/socrata/[id].json

```

1 >>> r = requests.delete("http://127.0.0.1:8000/
2   socrata/default/api/socrata/3.json")
3 >>> r
4 <Response [200]>
5 >>> r = requests.get("http://127.0.0.1:8000/
6   socrata/default/api/socrata/id/3/name.json")
7 >>> r
8 <Response [404]>
9 >>> r.content
10 '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.

```

1 auth.settings.allow_basic_login = True
2 @auth.requires_login()

```

Authorized:

```

1 >>> r =
   requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/1.json",
   auth=('michael@realpython.com', 'admin'))
2 >>> r.content
3 '{"content": [{"name": "2010 Report to Congress on
4   White House Staff", "views": "508,705",
5   "url": "https://opendata.socrata.com/Government/
6   2010-Report-to-Congress-on-White-House-Staff/vedg-c5sb", "id":
   1}]]\n'

```

Chapter 25

Django: Quick Start

Overview

Like web2py, Django 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. In spite of that, Django has a high learning curve due to much of the implicit automation that happens in the back-end. 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-weight/minimalist 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 implicit 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-Views architecture (MTV):

- **Models** represent your data model/tables, 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. This is the presentation layer and defines how information is displayed to the end user.



Figure 25.1: Django Logo

- **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 due to the automation of common web development tasks and integrated functionality (included batteries). As long as you are aware of the inherent structure and organization (scaffolding) 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 formula/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 application 1 shares code with application 2.

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 first [chapter](#) of the Django book.

Installation

1. Create a new directory called “django-quick-start”, then create and activate a virtualenv.
2. Now install Django 1.8:

```
1 $ pip install django==1.8.4
```

If you want to check the Django version currently installed within your virtualenv, open the Python shell and run the following commands:

```
1 >>> import django
2 >>> django.VERSION
3 (1, 8, 4, 'final', 0)
```

If you see a different version or an `ImportError`, make sure to uninstall Django, `pip uninstall Django`, then try installing Django again. Double-check that your virtualenv is activated before installing.

Hello, World!

As always, let's start with a basic Hello World app.

Basic Setup

1. Start a new Django project:

```
1 $ django-admin.py startproject hello_world_project
```

2. This command created the basic project layout, containing one directory and five files:

```
1
2 hello_world_project
3     __init__.py
4     settings.py
5     urls.py
6     wsgi.py
7 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 your 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.
3. 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 (Project root) and run the following command:

```
1 $ python manage.py runserver
```

You can specify a different port with the following command (if necessary):

```
1 $ python manage.py runserver 8080
```

You should see something similar to this:

```
1 Performing system checks...
2
3 System check identified no issues (0 silenced).
4
5 You have unapplied migrations; your app may not work properly until
  they are applied.
6 Run 'python manage.py migrate' to apply them.
7
8 September 05, 2015 - 14:41:37
9 Django version 1.8.4, using settings 'hello_world_project.settings'
10 Starting development server at http://127.0.0.1:8000/
11 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 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:

```
1 $ python manage.py syncdb
```

Don't set up a superuser.

Run the server again:

```
1 $ 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"

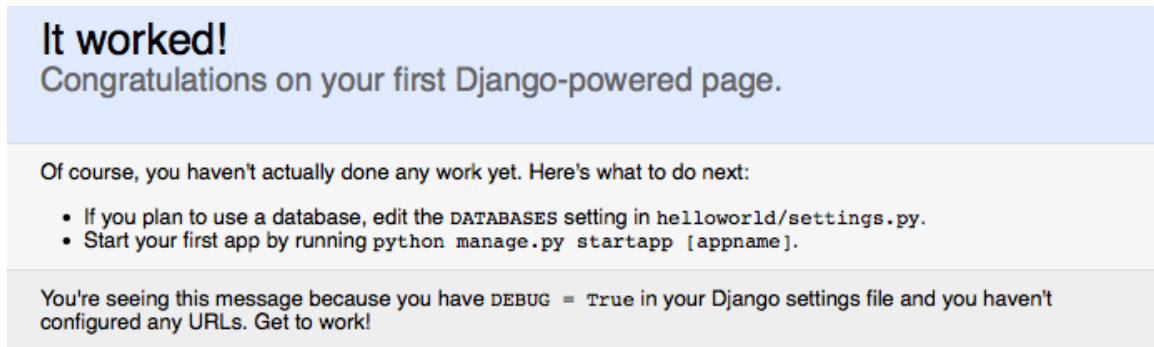


Figure 25.2: “Welcome to Django” screen

- 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

1. Now that the Django Project is setup, let’s create a new app. With virtualenv 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:** This is used to define your data models (entities and relationships) and map them to database table(s).
- **tests.py:** This 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 (as mentioned above), 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:

```

1
2 db.sqlite3
3 hello_world
4     __init__.py
5     admin.py
6     migrations
7     __init__.py
8     models.py
9     tests.py
10    views.py
11 hello_world_project
12     __init__.py
13     settings.py
14     urls.py
15     wsgi.py
16 manage.py

```

What's the difference between a Project and an App? A project is the main web app/site, 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 applications, you can theoretically reuse a Django application in a different Django project - so there's no need to reinvent the wheel.

2. 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.

Your “INSTALLED_APPS” section should look like this-

```
1 INSTALLED_APPS = (  
2     'django.contrib.admin',  
3     'django.contrib.auth',  
4     'django.contrib.contenttypes',  
5     'django.contrib.sessions',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'hello_world',  
9 )
```

URLs and Views

1. Open the *views.py* file and add the following code:

```
1 from django.http import HttpResponse  
2  
3  
4 def index(request):  
5     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, from a user, that triggered the view.
 - 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 “Hello, World!” to the browser.
2. 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:

```
1 from django.conf.urls import url  
2 from hello_world import views  
3
```



```

4
5 urlpatterns = [
6     url(r'^$', views.index, name='index'),
7 ]

```

3. 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:

```

1 from django.conf.urls import include, url
2 from django.contrib import admin
3
4 urlpatterns = [
5     url(r'^admin/', include(admin.site.urls)),
6     url(r'^hello/', include('hello_world.urls')),
7 ]

```

SEE ALSO: Please read the official Django documentation on [URLs](#) for further details and examples.

4. 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.
5. Navigate back to the root directory - <http://localhost:8000/>. You should see a 404 error, because we have not setup a view that maps to `/`.

Let's change that.

Open up your Project's *urls.py* file again and update the `urlpatterns` list:

```

1 from django.conf.urls import include, url
2 from django.contrib import admin
3
4 urlpatterns = [
5     url(r'^admin/', include(admin.site.urls)),
6     url(r'^hello/', include('hello_world.urls')),
7     url(r'^$', include('hello_world.urls')),
8 ]

```

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:

```
1 def about(request):
2     return HttpResponse(
3         "Here is the About Page. Want to return home? <a
4         href='/'>Back Home</a>"
5     )
```

url:

```
1 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.

1. Navigate to the root and create a new directory called "templates". Your project structure should now look like this:

```
1
2  hello_world
3      __init__.py
4      admin.py
5      migrations
6      __init__.py
7      models.py
8      tests.py
9      urls.py
10     views.py
11  hello_world_project
12     __init__.py
13     settings.py
14     urls.py
```

```

15     wsgi.py
16 manage.py
17 templates

```

2. Next, let's update the *settings.py* file to add the path to our “templates” directory to the list associated with the `DIRS` key so that your Django project knows where to find the templates:

```

1 'DIRS': [os.path.join(BASE_DIR, 'templates')],

```

3. Update the *views.py* file:

```

1 from django.template import Context, loader
2 from datetime import datetime
3 from django.http import HttpResponse
4
5
6 def index(request):
7     return HttpResponse('<html><body>Hello, World!</body></html>')
8
9
10 def about(request):
11     return HttpResponse(
12         "Here is the About Page. Want to return home? <a
13         href='/'>Back Home</a>"
14     )
15
16 def better(request):
17     t = loader.get_template('betterhello.html')
18     c = Context({'current_time': datetime.now(), })
19     return HttpResponse(t.render(c))

```

So in the `better()` function we use the `loader` method along with the `get_template()` function to return a template called `betterhello.html`. The `Context` class takes a dictionary of variable names, as the dict's keys, and their 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.

4. Next, let's setup the template. 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 `{ { }`. Place the following code in the file:

```

1 <html>
2 <head><title>A Better Hello!</title></head>
3 <body>
4   <p>Hello, World! This template was rendered on
      {{current_time}}.</p>
5 </body>
6 </html>

```

5. Finally, we need to add the view to the app's *urls.py* file:

```

1 url(r'^better/$', views.better, name='better'),

```

6. Fire up your server and navigate to <http://localhost:8000/better>. You should see something like this:

“Hello, World! This template was rendered on Sept. 5, 2015, 3:12 p.m..”

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'))`.

7. 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](http://en.wikipedia.org/wiki/List_of_tz_database_time_zones).

For example, if you change the time zone to `TIME_ZONE = 'America/Los_Angeles'`, and refresh <http://localhost:8000/better>, it should display the U.S. Pacific 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 Application...

Creating 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 Application

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.

NOTE: The above workflow is used in a number of Django tutorials and projects. Keep in mind, that there are a number of workflows you could follow/employ resulting in a number of different project structures.

We'll look at this workflow in greater detail in a later chapter.

Chapter 26

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: **JavaScript and jQuery**.

NOTE: Did you miss part 1? Jump back to **Interlude: Introduction to HTML and CSS**.

Along with HTML and CSS (which, again, provide structure and beauty, respectively) Javascript and jQuery are used to make webpages interactive.

Start by adding a *main.js* file to root directory and include the follow code:

```
1 $(function() {  
2     console.log("whee!")  
3 });
```

Then add the following files to your *index.html* just before the closing `</body>` tag:

```
1 <script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>  
2 <script src="http://netdna.bootstrapcdn.com/  
3     bootstrap/3.0.3/js/bootstrap.min.js"></script>  
4 <script src="main.js"></script>
```

Here we are just including jQuery, JavaScript, 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

- e.g. Firebug (Firefox) or Developer Tools (Chrome / Safari). If you've never used this before, Google "accessing the js console in Chrome" to learn how to pull up the JavaScript console.

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 libraries developed in JavaScript. It is possible to perform all the same functionality jQuery provides in vanilla JavaScript; it just takes a lot more code.

Update *main.js*:

```
1 $(function() {  
2  
3     console.log("whee!")  
4  
5     // event handler  
6     $("#btn-click").click(function() {  
7         if ($('#input').val() !== '') {  
8             var input = $("#input").val()  
9             console.log(input)  
10        }  
11    });  
12  
13 });
```

Add an id (`id="btn-click"`) to “index.html” within the `<button>` tags:

Before:

```
1 <button class="btn btn-primary btn-md">Submit!</button>
```

After:

```
1 <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/jQuery will not run until there is a button click.

2. `var input = $("input").val()` sets a variable called “input”, while `.val()` fetches the value from the form input.
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:

Append the text to 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? Quite 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.

main.js

Open up your JavaScript file and add this line of code:

```
1 $('ol').append('<li><a href="">x</a>' + input + '</li>');
```

Updated file:

```
1 $(function() {  
2  
3   console.log("whee!")  
4  
5   // event handler  
6   $("#btn-click").click(function() {  
7     if ($('#input').val() !== '') {  
8       // grab the value from the input box after the button click  
9       var input = $("#input").val()  
10      // display value within the browser's JS console  
11      console.log(input)  
12      // add the value to the DOM  
13      $('ol').append('<li><a href="">x</a>' + input + '</li>');  
14    }  
15    $('#input').val('');  
16  });  
17  
18 });
```

Then add the following code to your *index.html* file just below the button:

```
1 <br>  
2 <br>  
3 <h2>Todos</h2>  
4 <h3>  
5   <ol class="results"></ol>  
6 </h3>
```

What's going on?

`$('#ol').append('x' + input + '');` adds the value of the input variable to the DOM between the `<ol class="results">` and ``. Further, we're adding the value from the input, `input`, variable plus some HTML, `x - `. Also, the `$('#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.

Remove text from the DOM

Add the following code to *main.js*

```
1 $(document).on('click', 'a', function (e) {
2     e.preventDefault();
3     $(this).parent().remove();
4 });
```

The final code:

```
1 $(function() {
2
3     console.log("whee!")
4
5     // event handler
6     $("#btn-click").click(function() {
7         if ($('#input').val() !== '') {
8             // grab the value from the input box after the button click
9             var input = $("#input").val()
10            // display value within the browser's JS console
11            console.log(input)
12            // add the value to the DOM
13            $('ol').append('<li><a href="">x</a> - ' + input + '</li>');
14        }
15        $('#input').val('');
16    });
17
18 });
19
20 $(document).on('click', 'a', function (e) {
21     e.preventDefault();
22     $(this).parent().remove();
23 });
```

Here, on the event, the click of the link, we're removing that specific todo from the DOM. `e.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](#).
- See the Codecademy tracks on [JavaScript](#) and [jQuery](#) for more practice.
- Spend some time with dev tools. Read the official [documentation](#) for help.

Chapter 27

Bloggy: A blog app

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 your new Django Project:

```
1 $ mkdir django-bloggy
2 $ cd django-bloggy
3 $ virtualenv env
4 $ source env/bin/activate
5 $ pip install django==1.8.4
6 $ pip freeze > requirements.txt
7 $ django-admin.py startproject bloggy_project
```

Add the Django project folder to Sublime as a new project.

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*:

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         'NAME': os.path.join(BASE_DIR, 'bloggy.db'),  
5     }  
6 }
```

Sync the database

```
1 $ cd bloggy_project  
2 $ python manage.py migrate
```

This command creates the basic tables based on the apps in the `INSTALLED_APPS` tuple in your *settings.py* file. Did it ask you to setup a superuser? Use “admin” for both your username and password. 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 to ensure the new project is setup correctly:

```
1 $ 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

```
1 $ python manage.py startapp blog
```

Setup the model

Define the model for your application (e.g., what can be stored in the database) using the Django ORM (Object Relational Mapping) functions rather than with raw SQL. To do so, add the following code to *models.py*:

```
1 from django.db import models
2
3
4 class Post(models.Model):
5     created_at = models.DateTimeField(auto_now_add=True)
6     title = models.CharField(max_length=100)
7     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 using CRUD (create, read, update, and delete) 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 `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:

```
1 INSTALLED_APPS = (
2     # Django Apps
3     'django.contrib.admin',
```

```

4     'django.contrib.auth',
5     'django.contrib.contenttypes',
6     'django.contrib.sessions',
7     'django.contrib.messages',
8     'django.contrib.staticfiles',
9
10    # Local Apps
11    'blog',
12 )

```

Migrations

Execute the underlying SQL statements to create the database tables by creating then applying the [migration](#):

```
1 $ python manage.py makemigrations blog
```

You should see something similar to:

```

1 Migrations for 'blog':
2   0001_initial.py:
3     - Create model Post

```

The `makemigrations` command essentially tells Django that changes were 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:

```
1 $ python manage.py migrate
```

You should see something like:

```

1 Operations to perform:
2   Apply all migrations: admin, blog, contenttypes, auth, sessions
3 Running migrations:
4   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:

```
1 $ 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:

```
1 >>> python manage.py shell
2 >>> 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:

```
1 >>> Post.objects.all()
2 []
```

Adding data

To add new rows, we can use the following commands:

```
1 >>> 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")
2 >>> p.save()
3 >>> 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 'companys
success. Here is a look at five charting best practices for
optimal data visualization and analysis.")
4 >>> p.save()
5 >>> p = Post(title="Understand Your Support System Better With
Sentiment Analysis", content="'Theres 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 'youre on the verge of running
clients away.")
6 >>> p.save()

```

NOTE: Remember that we don't need to add a primary key, `id`, 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:

```

1 >>> Post.objects.all()
2 [<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:

```

1 def __unicode__(self):
2     return self.title

```

Your file should now look like this:

```
1 from django.db import models
2
3
4 class Post(models.Model):
5     created_at = models.DateTimeField(auto_now_add=True)
6     title = models.CharField(max_length=100)
7     content = models.TextField()
8
9     def __unicode__(self):
10         return self.title
```

This is a bit confusing, as Python Classes are usually returned with `__str__`, not `__unicode__`. We're using unicode because Django models use unicode by [default](#).

NOTE: Using Python 3? Stick with `__str__`.

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:

```
1 [<Post: What Am I Good At>, <Post: Charting Best Practices>, <Post:
   Understand Your Support System Better With Sentiment Analysis>]
```

This should be much easier to read and understand. We know there are three rows in the table, and we know their titles.

Depending on how much information you want returned, you could add all the fields to the *models.py* file:

```
1 def __unicode__(self):
2     return str(self.id) + " / " + str(self.created_at) + " / " +
       self.title + " / " + self.content + "\n"
```

Test this out. What does this return? Make sure to update this when you're done so it's just returning the title again:

```
1 def __unicode__(self):
2     return self.title
```

Open up your [SQLite Browser](#) to make sure the data was added correctly:

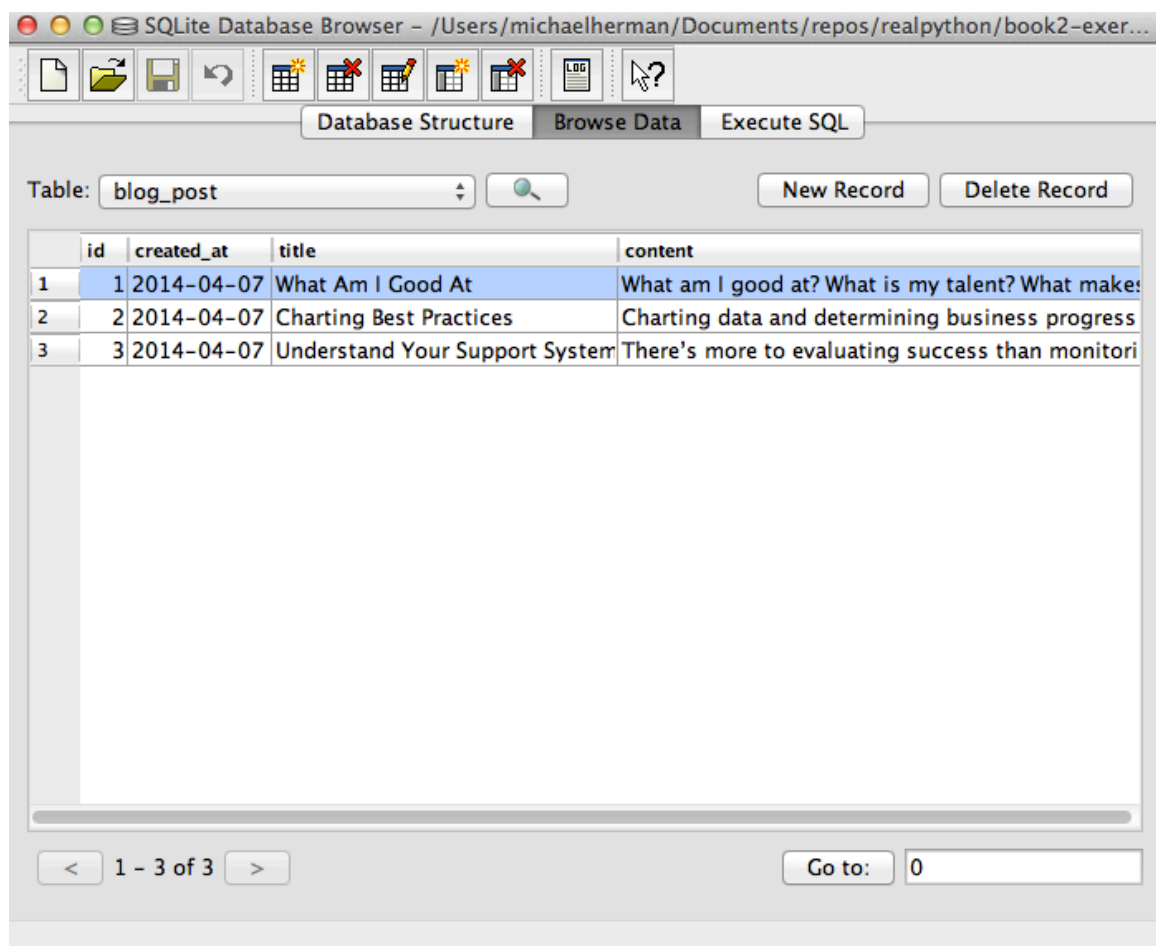


Figure 27.1: Verify data within SQLite Browser (Django)

Searching (slightly more advanced)

Let's look at how to query then filter the data from the Shell.

Import the Model:

```
1 >>> from blog.models import Post
```

Return objects by a specific id:

```
1 >>> Post.objects.filter(id=1)
```

Return objects by ids > 1:

```
1 >> Post.objects.filter(id__gt=1)
```

Return objects where the title contains a specific word:

```
1 >>> 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*:

```
1 from django.test import TestCase
2 from blog.models import Post
3
4
5 class PostTests(TestCase):
6
7     def test_str(self):
8         my_title = Post(title='This is a basic title for a basic
9                             test case')
10        self.assertEqual(
11            str(my_title), 'This is a basic title for a basic test
12                           case',
13        )
```

Run the test case:

```
1 $ 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:

```
1 reating test database for alias 'default' (':memory:')...
2 Operations to perform:
3   Synchronize unmigrated apps: staticfiles, messages
4   Apply all migrations: admin, blog, contenttypes, auth, sessions
5 Synchronizing apps without migrations:
6   Creating tables...
7   Running deferred SQL...
8   Installing custom SQL...
9 Running migrations:
```

```

10 Rendering model states... DONE
11 Applying contenttypes.0001_initial... OK
12 Applying auth.0001_initial... OK
13 Applying admin.0001_initial... OK
14 Applying contenttypes.0002_remove_content_type_name... OK
15 Applying auth.0002_alter_permission_name_max_length... OK
16 Applying auth.0003_alter_user_email_max_length... OK
17 Applying auth.0004_alter_user_username_opts... OK
18 Applying auth.0005_alter_user_last_login_null... OK
19 Applying auth.0006_require_contenttypes_0002... OK
20 Applying blog.0001_initial... OK
21 Applying sessions.0001_initial... OK
22 test_str (blog.tests.PostTests) ... ok
23
24 -----
25 Ran 1 test in 0.001s
26
27 OK
28 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:

```
1 from django.contrib import admin
2 from blog.models import Post
3
4 admin.site.register(Post)
```

Here, we're simply telling Django which models we want 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 (“admin” and “admin”).

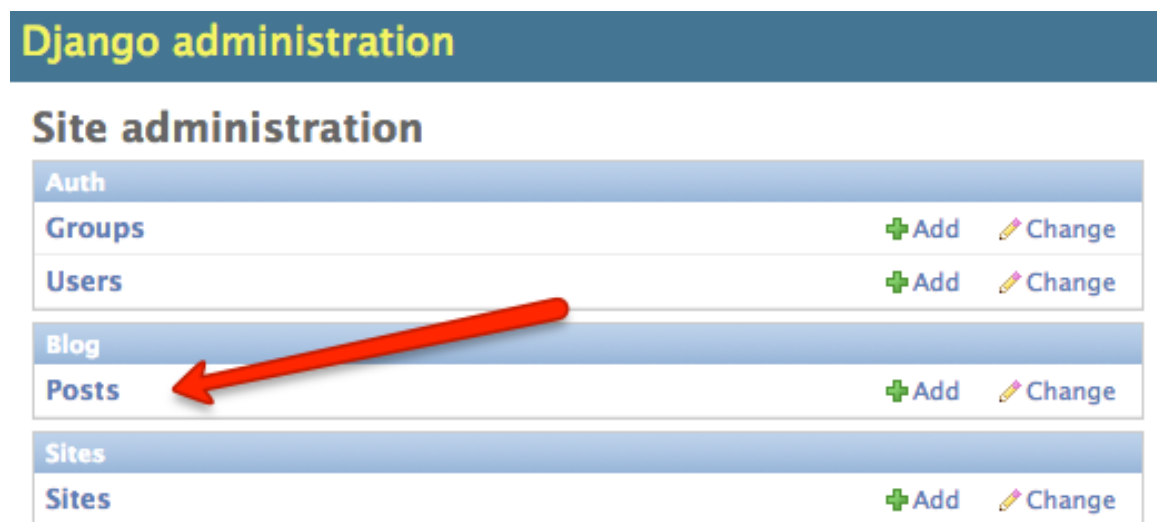


Figure 27.2: Django Admin

Add some more posts, change some posts, delete some posts. Go crazy.

Custom Admin View

We can customize the admin views by simply editing our *admin.py* files. 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:

```
1 class PostAdmin(admin.ModelAdmin):  
2     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:

```
1 class PostAdmin(admin.ModelAdmin):  
2     list_display = ('title', 'created_at')
```

Finally, register this class with with Django's admin interface:

```
1 admin.site.register(Post, PostAdmin)
```

Updated *admin.py* code:

```
1 from django.contrib import admin  
2 from blog.models import Post  
3  
4  
5 class PostAdmin(admin.ModelAdmin):  
6     list_display = ('title', 'created_at')  
7  
8 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:

```
1 'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

Your Project structure should now look like this:

```
1
2 bloggy_project
3     blog
4         __init__.py
5         admin.py
6         migrations
7         models.py
8         tests.py
9         views.py
10    bloggy.db
11    bloggy_project
12        __init__.py
13        settings.py
14        urls.py
15        wsgi.py
16    manage.py
17    templates
18        blog
```

View

Add the following code to the *views.py* file:

```
1 from django.http import HttpResponseRedirect
2 from django.template import Context, loader
3
4 from blog.models import Post
```

```

5
6
7 def index(request):
8     latest_posts = Post.objects.all().order_by('-created_at')
9     t = loader.get_template('blog/index.html')
10    c = Context({'latest_posts': latest_posts, })
11    return HttpResponse(t.render(c))

```

Template

Create a new file called *index.html* within the “templates/blog” directory and add the following code:

```

1 <h1>Bloggy: a blog app</h1>
2 {% if latest_posts %}
3 <ul>
4 {% for post in latest_posts %}
5   <li>{{post.created_at}} | {{post.title}} | {{post.content}}</li>
6 {% endfor %}
7 </ul>
8 {% endif %}

```

URL

Add a *urls.py* file to your “blog” directory, then add the following code:

```

1 from django.conf.urls import url
2 from blog import views
3
4
5 urlpatterns = [
6     url(r'^$', views.index, name='index'),
7 ]

```

Now update the Project’s *urls.py* file:

```

1 from django.conf.urls import include, url
2 from django.contrib import admin
3
4 urlpatterns = [
5     url(r'^admin/', include(admin.site.urls)),

```

```

6     url(r'^blog/', include('blog.urls')),
7 ]

```

Test

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

Figure 27.3: Basic Django Blog View

Refactor

Let's clean this up a bit.

Template (second iteration)

Update *index.html*:

```

1 <html>
2 <head>
3   <title>Bloggy: a blog app</title>
4 </head>
5 <body>
6 <h2>Welcome to Bloggy!</h2>
7 {% if latest_posts %}
8 <ul>
9 {% for post in latest_posts %}
10  <h3><a href="/blog/{{ post.id }}">{{ post.title }}</a></h3>

```

```

11 <p><em>{{ post.created_at }}</em></p>
12 <p>{{ post.content }}</p>
13 <br>
14 {% endfor %}
15 </ul>
16 {% endif %}
17 </body>
18 </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 routing or the template. Let's do that now.

Also, 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.

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:

```

1 from django.conf.urls import url
2 from blog import views
3
4
5 urlpatterns = [
6     url(r'^$', views.index, name='index'),
7     url(r'^(?P<post_id>\d+)/$', views.post, name='post'),
8 ]

```

Take a look at the new URL. The pattern, `(?P<slug>\d+)`, we used is made up of regular expressions. Click [here](#) to learn more about regular expressions. There's also a chapter on regular expressions within the first Real Python course. You can test out regular expressions using [Pythex](#) or the [Python Regular Expression Tool](#):

Python Regular Expression Testing Tool

Regular Expression (regex)
blog/(?P<post_id>\d+)

Flags
☐ Ignore case ☐ Locale ☐ Multi line ☐ Dot all ☐ Unicode ☐ Verbose

String
blog/1

Timeout
☐ Timeout loops 10000

Code:

```
>>> regex = re.compile("blog/(?P<post_id>\d+)")
>>> r = regex.search(string)
>>> r
<_sre.SRE_Match object at 0x4b2a4b0c45dc9950>
>>> regex.match(string)
<_sre.SRE_Match object at 0x4b2a4b0c45c51768>

# List the groups found
>>> r.groups()
(u'1',)

# List the named dictionary objects found
>>> r.groupdict()
{'post_id': u'1'}

# Run findall
>>> regex.findall(string)
[u'1']
```

Figure 27.4: Regular Express test in the Python Regular Expression Tool

View

Add a new function to *views.py*:

```
1 from django.shortcuts import get_object_or_404
2
3 def post(request, post_id):
4     single_post = get_object_or_404(Post, pk=post_id)
5     t = loader.get_template('blog/post.html')
6     c = Context({'single_post': single_post, })
7     return HttpResponse(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.

Template:

Create a new template called *post.html*:

```
1 <html>
2 <head>
3   <title>Bloggy: {{ single_post.title }}</title>
4 </head>
5 <body>
6 <h2>{{ single_post.title }}</h2>
7 <ul>
8   <p><em>{{ single_post.created_at }}</em></p>
9   <p>{{ single_post.content }}</p>
10  <br/>
11 </ul>
12 <p>Had enough? Return <a href="/blog">home</a>.</p><br/>
13 </body>
14 </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. 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:

```
1 def index(request):
2     latest_posts = Post.objects.all().order_by('-created_at')
3     t = loader.get_template('blog/index.html')
4     context_dict = {'latest_posts': latest_posts, }
5     for post in latest_posts:
6         post.url = post.title.replace(' ', '_')
7     c = Context(context_dict)
8     return HttpResponse(t.render(c))
```

The main difference is that we're creating a `post.url` using a `for` loop to replace the spaces in a post name with underscores:

```
1 for post in latest_posts:
2     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:

```
1 <h3><a href="/blog/{ post.post_id }">{{ post.title }}</a></h3>
```

With:

```
1 <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:

```
1 def post(request, post_url):
2     single_post = get_object_or_404(Post,
3         title=post_url.replace('_', ' '))
4     t = loader.get_template('blog/post.html')
5     c = Context({'single_post': single_post, })
6     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:

```
1 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 in the [Python Regular Expression Tool](#).

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:

```
1 tag = models.CharField(max_length=20, blank=True, null=True)
2 image = models.ImageField(upload_to="images", blank=True, null=True)
3 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):

```

1 django.core.management.base.SystemCheckError: SystemCheckError:
   System check identified some issues:
2
3 ERRORS:
4 blog.Post.image: (fields.E210) Cannot use ImageField because Pillow
   is not installed.
5     HINT: Get Pillow at https://pypi.python.org/pypi/Pillow or run
       command "pip install Pillow".
6
7 System check identified 1 issue (0 silenced).

```

Install Pillow:

```

1 $ pip install Pillow==2.9.0
2 $ pip freeze > requirements.txt

```

Fire up your development server and login 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:

```

1 $ python manage.py makemigrations

```

Output:

```

1 Migrations for 'blog':
2   0002_auto_20140918_2218.py:
3     - Add field image to post
4     - Add field tag to post
5     - 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 always want to double-check the migration file to ensure that it is correct. Complex changes may not always turn out the way you expected. You can edit the migration file directly, if necessary.

Apply migrations

Now, let's apply those migrations to the database:

```
1 $ python manage.py migrate
```

Output:

```
1 Operations to perform:
2   Synchronize unmigrated apps: staticfiles, messages
3   Apply all migrations: admin, blog, contenttypes, auth, sessions
4 Synchronizing apps without migrations:
5   Creating tables...
6   Running deferred SQL...
7   Installing custom SQL...
8 Running migrations:
9   Rendering model states... DONE
10  Applying blog.0002_auto_20150905_0934... OK
```

Did this work? Let's find out.

Run the server, go the Django admin, and you now should be able to add new rows that include tags, an image, and/or the number of views. Boom!

Add a few posts with the new fields. *Don't add any images just yet, though.* 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:

```
1 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*:

```

1 <html>
2 <head>
3   <title>Bloggy: {{ single_post.title }}</title>
4 </head>
5 <body>
6 <h2>{{ single_post.title }}</h2>
7 <ul>
8   <p><em>{{ single_post.created_at }}</em></p>
9   <p>{{ single_post.content }}</p>
10  <p>Tag: {{ single_post.tag }}</p>
11  <br>
12  <p>
13    {% if single_post.image %}
14      
15    {% endif %}
16  </p>
17 </ul>
18 <p>Had enough? Return <a href="/blog">home</a>.</p><br/>
19 </body>
20 </html>

```

Update *settings.py*:

```

1 MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
2 MEDIA_URL = '/media/'
3 STATIC_ROOT = os.path.join(BASE_DIR, 'static')
4 STATIC_URL = '/static/'

```

Make sure to create these directories as well in the root.

Update Project-level *urls.py*:

```

1 from django.conf.urls import include, url
2 from django.contrib import admin
3 from .settings import MEDIA_ROOT
4
5
6 urlpatterns = [
7     url(r'^admin/', include(admin.site.urls)),
8     url(r'^blog/', include('blog.urls')),
9     url(r'^media/(?P<path>.*)$', 'django.views.static.serve',

```



```
10     {'document_root': MEDIA_ROOT}),
11 ]
```

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:

```
1 def post(request, post_url):
2     single_post = get_object_or_404(Post,
3         title=post_url.replace('_', ' '))
4     single_post.views += 1 # increment the number of views
5     single_post.save()    # and save it
6     t = loader.get_template('blog/post.html')
7     c = Context({'single_post': single_post, })
8     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*:

```
1 <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`. *This is the parent file. Add the following code, and save it in the main templates* directory.*

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Bloggy: a blog app</title>
6     <meta name="viewport" content="width=device-width,
7       initial-scale=1.0">
8     <meta name="description" content="">
9     <meta name="author" content="">
10
11     <!-- Le styles -->
12     <link href="https://maxcdn.bootstrapcdn.com/bootswatch/3.3.5/
13       yeti/bootstrap.min.css" rel="stylesheet">
14     <style>
15       body {
16         padding-top: 60px; /* 60px to make the container go all the
17           way to the bottom of the topbar */
18       }
19     </style>
20     <script src="http://code.jquery.com/jquery-1.11.0.min.js"
21       type="text/javascript"></script>
22     {% block extrahead %}
23     {% endblock %}
24     <script type="text/javascript">
25       $(function(){
26         {% block jquery %}
27         {% endblock %}
28       });
```

```

26     </script>
27 </head>
28
29 <body>
30
31     <div class="navbar navbar-inverse navbar-fixed-top"
32         role="navigation">
33         <div class="container">
34             <div class="navbar-header">
35                 <button type="button" class="navbar-toggle"
36                     data-toggle="collapse" data-target=".navbar-collapse">
37                     <span class="sr-only">Toggle navigation</span>
38                     <span class="icon-bar"></span>
39                     <span class="icon-bar"></span>
40                     <span class="icon-bar"></span>
41                 </button>
42                 <a class="navbar-brand" href="/blog">Bloggy</a>
43             </div>
44             <div class="collapse navbar-collapse">
45                 <ul class="nav navbar-nav">
46                     <li><a href="#">Add Post</a></li>
47                 </ul>
48             </div><!--/.nav-collapse -->
49         </div>
50
51     <div id="messages">
52         {% if messages %}
53         {% for message in messages %}
54             <div class="alert alert-{{message.tags}}">
55                 <a class="close" data-dismiss="alert">×</a>
56                 {{message}}
57             </div>
58         {% endfor %}
59         {% endif %}
60     </div>
61
62     <div class="container">
63         {% block content %}
64         {% endblock %}

```

```

64     </div> <!-- /container -->
65 </body>
66 </html>

```

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*:

```

1 {% extends '_base.html' %}
2
3 {% block content %}
4 {% if latest_posts %}
5 <ul>
6     {% for post in latest_posts %}
7         <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
8         <p><em>{{ post.created_at }}</em></p>
9         <p>{{ post.content }}</p>
10        <br>
11    {% endfor %}
12 </ul>
13 {% endif %}
14 {% endblock %}

```

Update *post.html*:

```

1 {% extends '_base.html' %}
2
3 {% block content %}
4 <h2>{{ single_post.title }}</h2>
5 <ul>
6     <p><em>{{ single_post.created_at }}</em></p>
7     <p>{{ single_post.content }}</p>
8     <p>Tag: {{ single_post.tag }}</p>
9     <p>Views: {{ single_post.views }}</p>
10    <br>
11    <p>

```

```
12 {% if single_post.image %}
13     
14 {% endif %}
15 </p>
16 </ul>
17 <p>Had enough? Return <a href="/blog">home</a>.</p><br/>
18 {% endblock %}
```

Take a look at the results. Amazing what five minutes - and [Bootstrap 3](#) - 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`:

```
1 def index(request):
2     latest_posts = Post.objects.all().order_by('-created_at')
3     popular_posts = Post.objects.order_by('-views')[:5]
4     t = loader.get_template('blog/index.html')
5     context_dict = {
6         'latest_posts': latest_posts,
7         'popular_posts': popular_posts,
8     }
9     for post in latest_posts:
10         post.url = post.title.replace(' ', '_')
11     c = Context(context_dict)
12     return HttpResponse(t.render(c))
```

Template

Update the `index.html` template, so that it loops through the `popular_posts`:

```
1 <h4>Must See:</h4>
2 <ul>
3     {% for popular_post in popular_posts %}
4         <li><a href="/blog/{{ popular_post.title }}">{{
5             popular_post.title }}</a></li>
6     {% endfor %}
7 </ul>
```

Updated file:

```
1 {% extends '_base.html' %}
2
3 {% block content %}
4 {% if latest_posts %}
```

```

5 <ul>
6   {% for post in latest_posts %}
7     <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
8     <p><em>{{ post.created_at }}</em></p>
9     <p>{{ post.content }}</p>
10    <br>
11    {% endfor %}
12 </ul>
13 {% endif %}
14 <h4>Must See:</h4>
15 <ul>
16   {% for popular_post in popular_posts %}
17     <li><a href="/blog/{{ popular_post.title }}">{{
18       popular_post.title }}</a></li>
19   {% endfor %}
20 </ul>
21 {% endblock %}

```

Bootstrap

Next let's add the Bootstrap [Grid System](#) to our *index.html* template:

```

1 {% extends '_base.html' %}
2
3 {% block content %}
4 <div class="row">
5   <div class="col-md-8">
6     {% if latest_posts %}
7       <ul>
8         {% for post in latest_posts %}
9           <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
10          <p><em>{{ post.created_at }}</em></p>
11          <p>{{ post.content }}</p>
12          <br>
13          {% endfor %}
14        </ul>
15      {% endif %}
16    </div>
17    <div class="col-md-4">
18      <h4>Must See:</h4>

```

```

19     <ul>
20         {% for popular_post in popular_posts %}
21         <li><a href="/blog/{{ popular_post.title }}">{{
22             popular_post.title }}</a></li>
23         {% endfor %}
24     </ul>
25 </div>
26 {% endblock %}

```

For more info on how the Grid System works, please view [this](#) blog 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:

```

1 for popular_post in popular_posts:
2     popular_post.url = popular_post.title.replace(' ', '_')

```

Updated function:

```

1 def index(request):
2     latest_posts = Post.objects.all().order_by('-created_at')
3     popular_posts = Post.objects.order_by('-views')[:5]
4     t = loader.get_template('blog/index.html')
5     context_dict = {
6         'latest_posts': latest_posts,
7         'popular_posts': popular_posts,
8     }
9     for post in latest_posts:
10         post.url = post.title.replace(' ', '_')
11     for popular_post in popular_posts:
12         popular_post.url = popular_post.title.replace(' ', '_')
13     c = Context(context_dict)
14     return HttpResponse(t.render(c))

```

Then update the URL within the *index.html* file:

```

1 <li><a href="/blog/{{ popular_post.url }}">{{ popular_post.title
2     }}</a></li>

```


Test this out. It should work.

View

Finally, let's add a `encode_url()` helper function to *views.py* and refactor the `index()` function to clean up the code:

```
1 # helper function
2 def encode_url(url):
3     return url.replace(' ', '_')
4
5
6 def index(request):
7     latest_posts = Post.objects.all().order_by('-created_at')
8     popular_posts = Post.objects.order_by('-views')[:5]
9     t = loader.get_template('blog/index.html')
10    context_dict = {
11        'latest_posts': latest_posts, 'popular_posts':
            popular_posts,
12    }
13    for post in latest_posts:
14        post.url = encode_url(post.title)
15    for popular_post in popular_posts:
16        popular_post.url = encode_url(popular_post.title)
17    c = Context(context_dict)
18    return HttpResponse(t.render(c))
```

Here, we're calling the helper function, `encode_url()` from the `for` loops in the `index()` function. We now are not repeating ourself by using this code - `replace(' ', '_')` - twice in the `index()` function. This should also give us a performance boost, especially if we are working with thousands of posts. It also allows us to use that helper function over and over again, in case other functions need to use it as well.

Test this out in your development server to make sure nothing broke. Try running your test suite as well:

```
1 $ python manage.py test -v 2
```

Homework

- Update the HTML (Bootstrap Grid) in the *posts.html* file as well as the code in the views to show the popular posts.
- Notice how we have some duplicate code in both the `index()` and `post()` functions. This should trigger a [code smell](#). How can you write a helper function for this so that you're only writing that code once?

Forms

In this section, we'll look at how to add forms to our Blog to allow end-users to add posts utilizing Django's built-in [form handling](#) features.

Such features allow you 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; and,
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 `urlpatterns` for the new View

Let's get started.

Create a *forms.py* file to add fields to your form

Include the following code in *blog/forms.py*:

```

1 from django import forms
2 from blog.models import Post
3
4
5 class PostForm(forms.ModelForm):
6
7     class Meta:
8         model = Post
9         fields = ['title', 'content', 'tag', 'image', 'views']

```

Here we're creating a new `ModelForm` that's mapped to our model via the `Meta()` inner class `model = Post`. Notice how each of our form fields has an associated column in the database. This is required.

Add the handling of the form logic to the View

Next, let's update our app's 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*:

```

1 def add_post(request):
2     context = RequestContext(request)
3     if request.method == 'POST':
4         form = PostForm(request.POST, request.FILES)
5         if form.is_valid(): # is the form valid?
6             form.save(commit=True) # yes? save to database
7             return redirect(index)
8         else:
9             print form.errors # no? display errors to end user
10    else:
11        form = PostForm()
12    return render_to_response('blog/add_post.html', {'form': form},
        context)

```

Also be sure to update the imports:

```

1 from django.http import HttpResponseRedirect
2 from django.template import Context, loader, RequestContext
3 from django.shortcuts import get_object_or_404, render_to_response,
    redirect
4

```

```
5 from blog.models import Post
6 from blog.forms import PostForm
```

What's going on here?

1. First we determine if the request is a GET or POST. If the former, we are simply displaying the form, `form = PostForm()`; if the latter, we are going to process the form data:

```
1 form = PostForm(request.POST, request.FILES)
2 if form.is_valid(): # is the form valid?
3     form.save(commit=True) # yes? save to database
4     return redirect(index)
5 else:
6     print form.errors # no? display errors to end user
```

2. 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.

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.

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](#).

3. 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.

NOTE: Did you notice the `render_to_response()` function. We'll discuss this more in the final Django chapter.

Add or update a template to display the form

Moving on, let's create the template called `add_post.html` within our “templates/blog” directory:

```
1 {% extends '_base.html' %}
2
3 {% block content %}
4 <h2>Add a Post</h2>
5 <form id="post_form" method="post" action="/blog/add_post/"
6     enctype="multipart/form-data">
7     {% csrf_token %}
8     {% for hidden in form.hidden_fields %}
9         {{ hidden }}
10    {% endfor %}
11
12    {% for field in form.visible_fields %}
13        {{ field.errors }}
14        {{ label_tag }}
15        {{ field }}
16    {% endfor %}
17
18    <input type="submit" name="submit" value="Create Post">
19 </form>
20 {% 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 our 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:

```
1 {% extends '_base.html' %}
2
3 {% block content %}
```

```

4 <h2>Add a Post</h2>
5 <form id="post_form" method="post" action="/blog/add_post/"
  enctype="multipart/form-data">
6   {% csrf_token %}
7
8   {{ form }}
9
10  <input type="submit" name="submit" value="Create Post">
11 </form>
12 {% 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:

```

1 {% if form.is_multipart %}
2   <form enctype="multipart/form-data" method="post" action="">
3 {% else %}
4   <form method="post" action="">
5 {% endif %}
6 {{ form }}
7 </form>

```

Add a urlpattern for the new View

Now we just need to map the view, add_post, to a URL:

```

1 from django.conf.urls import url
2 from blog import views

```

```

3
4
5 urlpatterns = [
6     url(r'^$', views.index, name='index'),
7     url(r'^add_post/', views.add_post, name='add_post'), # add
                        post form
8     url(r'^(?P<slug>[\w|\-]+)/$', views.post, name='post'),
9 ]

```

That's it. Just update the 'Add Post' link in our *_base.html* template-

```

1 <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/.

The screenshot shows a web browser window with a dark header bar containing the text 'Bloggy' and 'Add Post'. Below the header, the main heading 'Add a Post' is displayed. The form consists of a large, empty text area for the post content. At the bottom of the form, there are four input fields: 'Title', 'Content', 'Tag', and 'Image'. The 'Content' field is a large text area. Below the 'Title' field, there is a 'Choose File' button and the text 'No file chosen'. At the bottom left of the form, there is a 'Create Post' button.

Figure 27.5: Basic Django Blog Form View

Looks pretty bad. We can clean this up quickly with Bootstrap, specifically with the [Django Forms Bootstrap](#) package.

Setup is simple.

Install the package

```
1 $ pip install django-forms-bootstrap==3.0.0
2 $ pip freeze > requirements.txt
```

Update *settings.py*

Add the package to your INSTALLED_APPS tuple:

```
1 INSTALLED_APPS = (
2     # Django Apps
3     'django.contrib.admin',
4     'django.contrib.auth',
5     'django.contrib.contenttypes',
6     'django.contrib.sessions',
7     'django.contrib.messages',
8     'django.contrib.staticfiles',
9
10    # Third Party Apps
11    'django_forms_bootstrap',
12
13    # Local Apps
14    'blog',
15 )
```

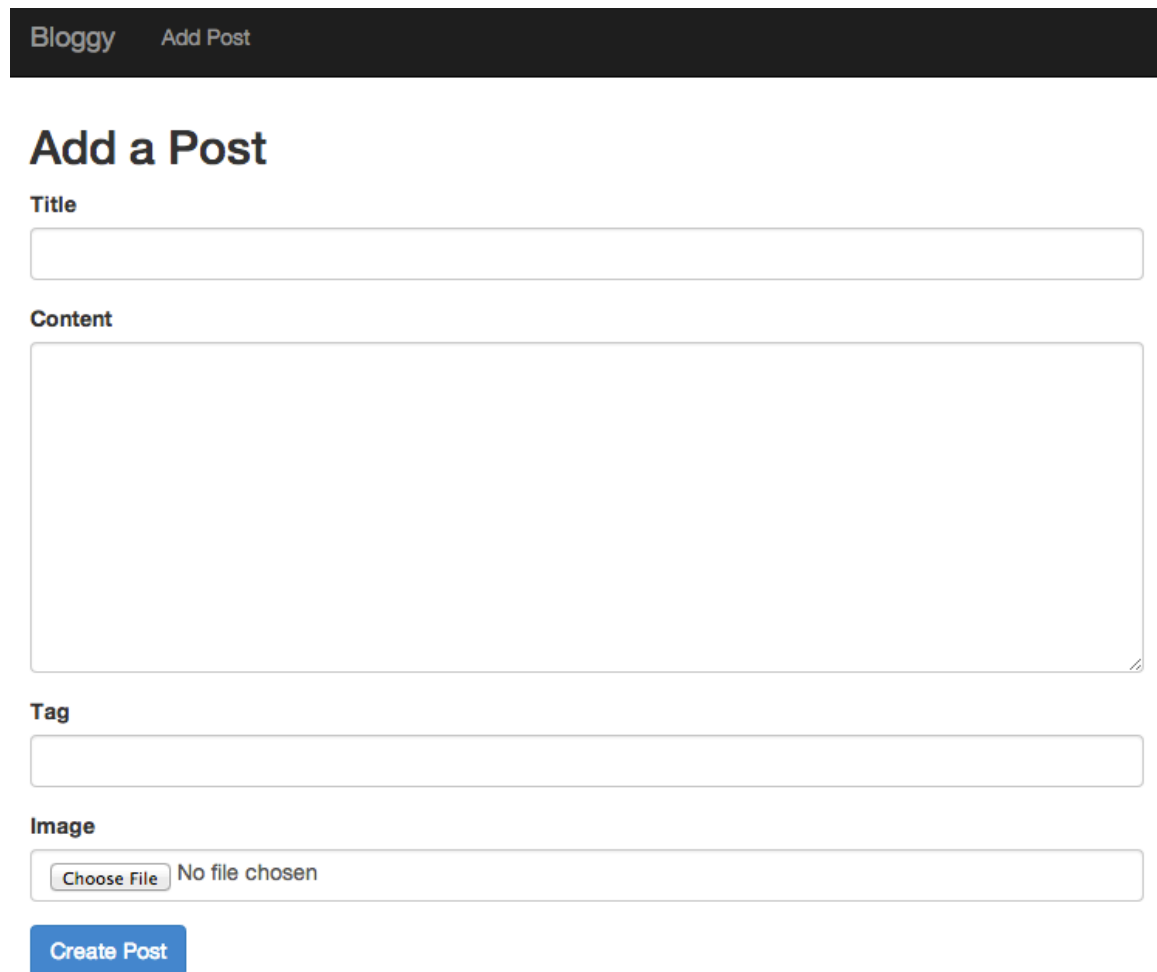
Update the template

```
1 {% extends '_base.html' %}
2
3 {% block content %}
4 {% load bootstrap_tags %}
5 <h2>Add a Post</h2>
6 <form id="post_form" method="post" action="/blog/add_post/"
7     enctype="multipart/form-data">
8     {% csrf_token %}
9
10    {{ form | as_bootstrap }}
11
12    <input type="submit" name="submit" value="Create Post" class="btn
13        btn-primary">
```

```
12 </form>
13 <br>
14 {% endblock %}
```

Sanity check

Restart the server. Check out the results:



The screenshot shows a web application interface for adding a blog post. At the top is a dark navigation bar with the text 'Bloggy' and a link 'Add Post'. Below this is a section titled 'Add a Post'. It contains four form fields: 'Title' (a single-line text input), 'Content' (a large multi-line text area), 'Tag' (a single-line text input), and 'Image' (a file upload field with a 'Choose File' button and the text 'No file chosen'). At the bottom of the form is a blue button labeled 'Create Post'.

Figure 27.6: Basic Django Blog Form View (with custom styles)

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*:

```
1 from django import forms
2 from blog.models import Post
3
4
5 class PostForm(forms.ModelForm):
6
7     class Meta:
8         model = Post
9         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:

```
1 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

```
1 $ pip install django-uuslug==1.0.2
2 $ pip freeze > requirements.txt
```

Model

Update *models.py*:

```
1 from django.db import models
2
3 from uuslug import uuslug
4
5
6 class Post(models.Model):
7     created_at = models.DateTimeField(auto_now_add=True)
8     title = models.CharField(max_length=100)
9     content = models.TextField()
10    tag = models.CharField(max_length=20, blank=True, null=True)
11    image = models.ImageField(upload_to="images", blank=True,
12                               null=True)
13    views = models.IntegerField(default=0)
14    slug = models.CharField(max_length=100, unique=True)
15
16    def __unicode__(self):
17        return self.title
18
19    def save(self, *args, **kwargs):
20        self.slug = uuslug(self.title, instance=self,
21                           max_length=100)
22        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 Unslug documentation](#).

What's next?

```
1 $ python manage.py makemigrations
```

However, it's not that easy this time. When you run that command, you'll see:

```
1 You are trying to add a non-nullable field 'slug' to post without a
  default;
2 we can't do that (the database needs something to populate existing
  rows).
3 Please select a fix:
4 1) Provide a one-off default now (will be set on all existing rows)
5 2) Quit, and let me add a default in models.py
6 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.

```
1 Select an option: 1
2 Please enter the default value now, as valid Python
3 The datetime module is available, so you can do e.g.
  datetime.date.today()
4 >>> "test"
5 Migrations for 'blog':
6 0003_post_slug.py:
7   - Add field slug to post
```

Before we can migrate this, we have another problem. Take a look at the new field we're adding:

```
1 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:

```
1 class Migration(migrations.Migration):
2
3     dependencies = [
```

```

4         ('blog', '0002_auto_20140918_2218'),
5     ]
6
7     operations = [
8         migrations.AddField(
9             model_name='post',
10            name='slug',
11            field=models.CharField(default='test', max_length=100),
12            preserve_default=False,
13        ),
14    ]

```

NOTE: 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*:

```

1 slug = models.CharField(max_length=100)

```

Push through the migration:

```

1 $ python manage.py migrate
2 Operations to perform:
3   Synchronize unmigrated apps: django_forms_bootstrap
4   Apply all migrations: admin, blog, contenttypes, auth, sessions
5 Synchronizing apps without migrations:
6   Creating tables...
7   Installing custom SQL...
8   Installing indexes...
9 Running migrations:
10   Applying blog.0003_post_slug... OK

```

Now, let's update those fields to make them unique.

Django Shell

Start the Shell:

```

1 $ python manage.py shell

```

Grab all the posts, then loop through them appending the post title to a list:

```

1 >>> from blog.models import Post
2 >>> all_posts = Post.objects.all()
3 >>> titles = []
4 >>> for post in all_posts:
5 ...     titles.append(post.title)
6 ...
7 >>> print titles
8 [u'test', u'Charting Best Practices', u'Understand Your Support
   System Better With Sentiment Analysis', u'testing', u'pic
   test', u'look at this', u'hey']

```

Now, we need to update the slug field like so:

```

1 >>> from uuslug import slugify
2 >>> x = 1
3 >>> y = 0
4 >>> len(titles)
5 8
6 >>> while x <= 8:
7 ...     Post.objects.filter(id=x).update(slug=slugify(titles[y]))
8 ...     x += 1
9 ...     y += 1
10 ...
11 1
12 1
13 1
14 1
15 1
16 1
17 1
18 >>>

```

Exit the shell.

Model

Update the slug field in the model:

```

1 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*:

From:

```
1 url(r'^(?P<post_url>\w+)/$', views.post, name='post'),
```

To:

```
1 url(r'^(?P<slug>[\w|\-]+)/$', views.post, name='post'),
```

Here we're matching 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](#) or the [Python Regular Expression Tool](#).

View

Now we can simplify the *views.py* file (making sure to replace *post_url* with *slug*):

```
1 from django.http import HttpResponse
2 from django.template import Context, loader, RequestContext
3 from django.shortcuts import get_object_or_404, render_to_response,
  redirect
4
5 from blog.models import Post
6 from blog.forms import PostForm
7
8
9 #####
10 # helper functions #
11 #####
12
13 def get_popular_posts():
14     popular_posts = Post.objects.order_by('-views')[:5]
15     return popular_posts
16
17
18 #####
19 # view functions #
20 #####
```



```

21
22 def index(request):
23     latest_posts = Post.objects.all().order_by('-created_at')
24     t = loader.get_template('blog/index.html')
25     context_dict = {
26         'latest_posts': latest_posts,
27         'popular_posts': get_popular_posts(),
28     }
29     c = Context(context_dict)
30     return HttpResponse(t.render(c))
31
32
33 def post(request, slug):
34     single_post = get_object_or_404(Post, slug=slug)
35     single_post.views += 1 # increment the number of views
36     single_post.save() # and save it
37     t = loader.get_template('blog/post.html')
38     context_dict = {
39         'single_post': single_post,
40         'popular_posts': get_popular_posts(),
41     }
42     c = Context(context_dict)
43     return HttpResponse(t.render(c))
44
45
46 def add_post(request):
47     context = RequestContext(request)
48     if request.method == 'POST':
49         form = PostForm(request.POST, request.FILES)
50         if form.is_valid(): # is the form valid?
51             form.save(commit=True) # yes? save to database
52             return redirect(index)
53         else:
54             print form.errors # no? display errors to end user
55     else:
56         form = PostForm()
57     return render_to_response('blog/add_post.html', {'form': form},
        context)

```

Can you spot the big difference?

Template

Now update the actual URL in the *index.html* template...

Replace:

```
1 <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
```

With:

```
1 <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:

```
1 <li><a href="/blog/{{ popular_post.url }}">{{ popular_post.title }}</a></li>
```

With:

```
1 <li><a href="/blog/{{ popular_post.slug }}">{{ popular_post.title }}</a></li>
```

Test

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 Login/Authentication? *Tests!* Add whatever you feel like. Find tutorials on the web for assistance. Show us the results.

Homework

- Go over the Django Quick-start at <https://realpython.com/learn/start-django/>. At the very least, go over Django 1.5 and 1.8. What are the differences between 1.5 and 1.7.8 with regard to the setup?
- 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.

Chapter 28

Django Workflow

The following is a basic workflow that you can use as a quick reference for developing a Django 1.8 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 virtualenv.
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 model (database) to get the required data back 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 to *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 at the App to define form-related classes; define your `ModelForm` classes here.
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 `urlpatterns` 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 inherent from the base template.

Chapter 29

Bloggy Redux: Introducing Blongo

Let's build a blog. Again. This time powered by 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, please 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:

```
1 $ mongo --version
2 MongoDB shell version: 3.0.4
```

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 out 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.

1. Create a new directory called “django-blongo”, enter the newly created directory, activate a virtualenv, and then install MongoEngine and PyMongo:

```
1 $ pip install mongoengine==0.8.8
2 $ pip install pymongo==2.7.2
3 $ pip freeze > requirements.txt
```

2. Now, let's test the install in the Python Shell:

```
1 >>> import pymongo
2 >>> client = pymongo.MongoClient("localhost", 27017)
3 >>> db = client.test
4 >>> db.name
5 u'test'
6 >>> db.my_collection
7 Collection(Database(MongoClient('localhost', 27017), u'test'),
8   u'my_collection')
9 >>> db.my_collection.save({"django": 10})
10 ObjectId('53e2e5a03386b7202c8bde61')
11 >>> db.my_collection.save({"flask": 20})
12 ObjectId('53e2e5b13386b7202c8bde62')
13 >>> db.my_collection.save({"web2py": 30})
14 ObjectId('53e2e5bc3386b7202c8bde63')
15 >>> db.my_collection
16 Collection(Database(MongoClient('localhost', 27017), u'test'),
17   u'my_collection')
18 >>> db.my_collection.find()
19 <pymongo.cursor.Cursor object at 0x10124f810>
20 >>> for item in db.my_collection.find():
21 ...     print item
22 ...
23 {u'_id': ObjectId('53e2e5a03386b7202c8bde61'), u'django': 10}
24 {u'flask': 20, u'_id': ObjectId('53e2e5b13386b7202c8bde62')}
25 {u'web2py': 30, u'_id': ObjectId('53e2e5bc3386b7202c8bde63')}
```

Here, we created a new database called, `test`, 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.

3. Test Script:

```
1 import pymongo
2
3 # Open the MongoDB connection
4 conn = pymongo.MongoClient('mongodb://localhost:27017')
5
6 # Print the available MongoDB databases
7 databases = conn.database_names()
8 for database in databases:
9     print database
10
11 # Close the MongoDB connection
12 conn.close()
```

Save this as *mongo.py*. This simply connects to your instance of Mongo, then outputs all collection names. It will come in handy.

Django Setup

1. Install Django within “django-blongo”, then create a new Project:

```
1 $ pip install django==1.8.4
2 $ django-admin.py startproject blongo_project
3 $ cd blongo_project/
```

2. Update *settings.py*:

```
1 # DATABASES = {
2 #     'default': {
3 #         'ENGINE': 'django.db.backends.sqlite3',
4 #         'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
5 #     }
6 # }
7
8 AUTHENTICATION_BACKENDS = (
9     'mongoengine.django.auth.MongoEngineBackend',
10 )
11
12 SESSION_ENGINE = 'mongoengine.django.sessions'
13
14 MONGO_DATABASE_NAME = 'blongo'
15
16 from mongoengine import connect
17 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

1. Start a new app:

```
1 $ python manage.py startapp blongo
```

2. Add the new app to the *settings.py* file:

```
1 INSTALLED_APPS = (  
2     'django.contrib.admin',  
3     'django.contrib.auth',  
4     'django.contrib.contenttypes',  
5     'django.contrib.sessions',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'blongo',  
9 )
```

3. 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:

```
1 from mongoengine import Document, StringField, DateTimeField  
2 import datetime  
3  
4  
5 class Post(Document):  
6     title = StringField(max_length=200, required=True)  
7     content = StringField(required=True)  
8     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:

```
1 $ python manage.py shell
2 >>> from blongo.models import Post
3 >>> import datetime
4 >>> Post(title='just a test', content='again, just a test',
          date_published=datetime.datetime.now).save()
5 <Post: Post object>
```

To access the data we can use the objects attribute from the Document() class:

```
1 >>> for post in Post.objects:
2 ...     print post.title
3 ...
4 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

```
1 from django.conf.urls import include, url
2 from django.contrib import admin
3
4
5 urlpatterns = [
6     url(r'^admin/', include(admin.site.urls)),
7     url(r'^$', include('blongo.urls')),
8 ]
```

App URLs

```
1 from django.conf.urls import url
2 from blongo import views
3
4
5 urlpatterns = [
6     url(r'^$', views.index, name='index'),
7 ]
```

Views

```
1 from django.http import HttpResponse
2 from django.template import Context, loader
3
4 from blongo.models import Post
5
6
7 def index(request):
8     latest_posts = Post.objects
9     t = loader.get_template('index.html')
10    context_dict = {'latest_posts': latest_posts}
11    c = Context(context_dict)
12    return HttpResponse(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*:

```
1  TEMPLATES = [  
2      {  
3          'BACKEND':  
4              'django.template.backends.django.DjangoTemplates',  
5          'DIRS': [os.path.join(BASE_DIR, 'templates')],  
6          'APP_DIRS': True,  
7          'OPTIONS': {  
8              'context_processors': [  
9                  'django.template.context_processors.debug',  
10                 'django.template.context_processors.request',  
11                 'django.contrib.auth.context_processors.auth',  
12                 'django.contrib.messages.context_processors.messages',  
13             ],  
14         },  
15     ],  
16 ]
```

Then add the directory to the Project:

```
1  .  
2  blongo  
3      __init__.py  
4      admin.py  
5      migrations  
6      __init__.py  
7      models.py  
8      tests.py  
9      views.py  
10 blongo_project  
11     __init__.py  
12     settings.py  
13     urls.py  
14     wsgi.py  
15 manage.py  
16 templates
```

Template

Finally add the template (*index.html):

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Blongo: Django + Mongo</title>
6     <meta name="viewport" content="width=device-width,
7       initial-scale=1.0">
8     <meta name="description" content="">
9     <meta name="author" content="">
10
11     <!-- Le styles -->
12     <link
13       href="https://maxcdn.bootstrapcdn.com/bootswatch/3.3.5/yeti/
14         bootstrap.min.css" rel="stylesheet">
15     <style>
16       body {
17         padding-top: 60px;
18       }
19     </style>
20     <script src="http://code.jquery.com/jquery-1.11.0.min.js"
21       type="text/javascript">
22       {% block extrahead %}
23       {% endblock %}
24     </script>
25   </head>
26
27   <body>
28
29     <div class="navbar navbar-inverse navbar-fixed-top"
30       role="navigation">
31       <div class="container">
32         <div class="navbar-header">
33           <button type="button" class="navbar-toggle"
34             data-toggle="collapse" data-target=".navbar-collapse">
35             <span class="sr-only">Toggle navigation</span>
36             <span class="icon-bar"></span>
```



```

32         <span class="icon-bar"></span>
33         <span class="icon-bar"></span>
34     </button>
35     <a class="navbar-brand" href="/">Blongo</a>
36 </div>
37 <div class="collapse navbar-collapse">
38     <ul class="nav navbar-nav">
39     </ul>
40 </div><!--/.nav-collapse -->
41 </div>
42 </div>
43
44 <div class="container">
45     {% if latest_posts %}
46     {% for post in latest_posts %}
47         <h3>{{ post.title }}</h3>
48         <p><em>{{ post.created_at }}</em></p>
49         <p>{{ post.content }}</p>
50         <br>
51     {% endfor %}
52     {% endif %}
53 </div> <!-- /container -->
54
55 </body>
56 </html>

```

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:

```
1 import pymongo
2
3
4 def get_collection(conn):
5
6     databases = conn.database_names()
7
8     if 'blongo' in databases:
9         # connect to the blongo database
10         db = conn.blongo
11         # grab all collections
12         collections = db.collection_names()
13         # output all collection
14         print "Collections"
15         print "-----"
16         for collection in collections:
17             print collection
18         # pick a collection to view
19         col = raw_input(
20             '\nInput a collection name to show its field names: '
21         )
22         if col in collections:
23             get_items(db[col].find())
24         else:
25             print "Sorry. The '{}'.collection does not
26                 exist.".format(col)
27     else:
28         print "Sorry. The 'blongo' database does not exist."
29
30     # Close the MongoDB connection
31     conn.close()
32
33 def get_items(collection_name):
34
35     for items in collection_name:
```

```
36         print items
37
38 if __name__ == '__main__':
39     # Open the MongoDB connection
40     conn = pymongo.Connection('mongodb://localhost:27017')
41     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 problems for you in the next course. Cheers!

Chapter 30

Django: Ecommerce Site

Overview

In the past twenty or so chapters 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 now it's time to put it all 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 project.

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 in the next course, *Advanced Web Development with Django*, into an enterprise-level application. The point of this chapter along with the next course is simple: We'll show you roughly the steps of what a start-up experiences as it experiences growth.

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 your potential users demand.

As you have seen, development is broken into logical chunks. If we 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 be restricting. *Do not stifle your creativity in the early stages.*

As you define each function and apply a design, put yourself in the end users' shoes. What can he see? What can she do? Do they really care? For example, if one of your application's functions 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 functions 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 s/he is taken to a new page, create that new page. Again, for each main function, answer these three questions:

1. What can he see?
2. What can she 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 website. Create a story board, and then after plenty of iterations, build a high-fidelity prototype. One of the quickest means of doing this is via the front-end framework, 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:

```
1 As a <role>
2 I want <goal>
3 In order to <benefit>
```

For example:

```
1 As a vistor
2 I want to sign up
3 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, but it is planned for the future. In fact, the entire BDD process is planned for the future. 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.

So, let's get a basic Django Project and App setup, still within the context of rapid development.

NOTE In most 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 moving 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.

Let's get our hands dirty!

Setup your Project and App

NOTE: We are purposely creating this project using an older version of Django, 1.5.5, and Bootstrap, 2.3.2. Be sure to use Python 2.7x as well. In the next course you will learn how to take this project structure and upgrade it to the latest versions of both Django and Bootstrap as well as to Python 3. Again this is intentional, as it's important to understand the process of upgrading. Further, the setup process for a Django 1.5 app is much different Django 1.8. It's good to understand both, since the majority of Django applications use versions older than 1.6.

Basic Setup

1. Create a new directory called “django_ecommerce”. Create and activate a virtualenv with that directory.

2. Install Django:

```
1 $ pip install django==1.5.8
```

3. Start a new Django project:

```
1 $ django-admin.py startproject django_ecommerce
```

4. Add your Django project folder to Sublime as a new project.

5. Setup a new app:

```
1 $ cd django_ecommerce
2 $ python manage.py startapp main
```

NOTE: Due to the various functions 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:

```
1 .
2  django_ecommerce
3      __init__.py
4      settings.py
```

```

5     urls.py
6     wsgi.py
7 main
8     __init__.py
9     models.py
10    tests.py
11    views.py
12 manage.py

```

6. Add your app to the `INSTALLED_APPS` section of *settings.py* as well as your project's absolute paths (add this to the top of *settings.py*):

```

1 INSTALLED_APPS = (
2     'django.contrib.auth',
3     'django.contrib.contenttypes',
4     'django.contrib.sessions',
5     'django.contrib.sites',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'main',

```

and

```

1 import os
2
3 PROJECT_ROOT = os.path.realpath(os.path.dirname(__file__))
4 SITE_ROOT = os.path.dirname(PROJECT_ROOT)

```

7. Create a new directory within the root directory called “templates”, and then add the absolute path to the *settings.py* file:

```

1 TEMPLATE_DIRS = (os.path.join(SITE_ROOT, 'templates'),)

```

8. Configure the admin page.

Uncomment the following url pattern in *settings.py*:

```

1 'django.contrib.admin',

```

Uncomment these three lines in *urls.py*:

```

1 from django.contrib import admin
2 admin.autodiscover()
3
4 url(r'^admin/', include(admin.site.urls)),

```

WARNING: Remember: Your development environment should be isolated from the rest of your development machine (via virtualenv) 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.

Add a landing page

Now that the main app is up, let's quickly add a bare bones landing page.

1. Add the following code to the *main* app's *views.py* file:

```
1 from django.shortcuts import render_to_response
2 from django.template import RequestContext
3
4
5 def index(request):
6     return render_to_response(
7         'index.html', context_instance=RequestContext(request)
8     )
```

This function, `index()`, takes a parameter, `request`, which is an object that has information 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_to_response()` [function](#)? This is simply used to render the given template.

2. Next, we need to add a new pattern to our Project's *urls.py*:

```
1 url(r'^$', 'main.views.index', name='home'),
```

3. Finally, we need to create the *index.html* template. First create a new file called *base.html* (the parent template) within the templates directory, and add the following code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Your MVP</title>
6     <meta name="viewport" content="width=device-width,
7       initial-scale=1.0">
8     <meta name="description" content="">
9     <meta name="author" content="">
```

```

9
10 <!-- Le styles -->
11 <link
12     href="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/
13     bootstrap-combined.min.css" rel="stylesheet">
14 <style>
15
16     body {
17         padding-top: 20px;
18         padding-bottom: 40px;
19     }
20
21     /* Custom container */
22     .container-narrow {
23         margin: 0 auto;
24         max-width: 700px;
25     }
26     .container-narrow > hr {
27         margin: 30px 0;
28     }
29
30     /* Main marketing message and sign up button */
31     .jumbotron {
32         margin: 60px 0;
33         text-align: center;
34     }
35     .jumbotron h1 {
36         font-size: 72px;
37         line-height: 1;
38     }
39     .jumbotron .btn {
40         font-size: 21px;
41         padding: 14px 24px;
42     }
43
44     /* Supporting marketing content */
45     .marketing {
46         margin: 60px 0;
47     }
48     .marketing p + h4 {

```

```

48     margin-top: 28px;
49 }
50
51 </style>
52 <script src="http://code.jquery.com/jquery-1.11.0.min.js"
53     type="text/javascript"></script>
54     {% block extrahead %}
55     {% endblock %}
56 <script type="text/javascript">
57     $(function(){
58         {% block jquery %}
59         {% endblock %}
60     });
61 </script>
62 </head>
63
64 <body>
65
66     <div class="container-narrow">
67
68         <div class="masthead">
69             <ul class="nav nav-pills pull-right">
70                 <li><a href="{% url 'home' %}">Home</a></li>
71                 <li><a href="#">About</a></li>
72                 <li><a href="#">Contact</a></li>
73             </ul>
74             <h3><span class="fui-settings-16 muted">Your MVP!</span></h3>
75         </div>
76
77         <hr>
78
79         {% if messages %}
80         <div class="alert alert-success">
81             <div class="messages">
82                 {% for message in messages %}
83                 {{ message }}
84             {% endfor %}
85         </div>
86     </div>
87     {% endif %}

```

```

87
88     <div class="container-narrow">
89         {% block content %}
90         {% endblock %}
91     </div>
92
93     <hr>
94
95     <div class="footer">
96         <p>Copyright © 2014 <a href="http://www.yoursite.com">Your
          MVP</a>. Developed by <a
          href="http://realpython.com">Your Name Here</a>. Powered
          by Django.</P></p>
97     </div>
98
99 </div>
100
101 </body>
102 </html>

```

Next, add the *index.html* (child) template:

```

1 {% extends 'base.html' %}
2
3 {% block content %}
4
5 <div class="jumbotron">
6     
8     <h1>Please put some text here.</h1>
9     <p class="lead">If you want, you can add some text here as well.
10     Or not.</p>
11     <a class="btn btn-large btn-success" href="contact">Contact us
12     today to get started</a><br/><br/><br/>
13     <a href="https://twitter.com/RealPython"
14     class="twitter-follow-button" data-show-count="false"
15     data-size="large">Follow @RealPython</a>
16
17 <script>
18     !function(d,s,id){
19         var js,fjs=d.getElementsByTagName(s)[0],
20         p=/^http:/.test(d.location)?

```



```

50 <div class="row-fluid marketing">
51   <div class="span6">
52     <h3>(1) One</h3>
53     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
54     <h3>(2) Two</h3>
55     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
56     <h3>(3) Three</h3>
57     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
58     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
59   </div>
60
61   <div class="span6">
62     <h3>(4) Four</h3>
63     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
64     <h3>(5) Five</h3>
65     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
66     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
67     <h3>(6) Six</h3>
68     <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.

```

```

        Mauris ut dui ac nisi vestibulum accumsan. Praesent gravida
        nulla vitae arcu blandit, non congue elit tempus.
        Suspendisse quis vestibulum diam.</p>
69     </p>
70   </div>
71
72 </div>
73
74 {% endblock %}

```

By now you should understand the relationship between the parent and child templates; the relationship is the same for Django templates as it is for Jinja templates in Flask.

Also, your project structure should now look like this:

```

1  .
2  django_ecommerce
3      __init__.py
4      settings.py
5      urls.py
6      wsgi.py
7  main
8      __init__.py
9      models.py
10     tests.py
11     views.py
12  manage.py
13  templates
14     base.html
15     index.html

```

4. Fire up your server:

```

1  $ 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.

Your MVP!

[Home](#)

[About](#)

[Contact](#)



Please put some text here.

If you want, you can add some text here as well. Or not.

[Contact us today to get started](#)

Figure 30.1: Django Ecommerce MVP Example 1

Bootstrap

We used the [Bootstrap](#) front end framework to quickly add a basic design. To avoid looking too much like just 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, and potential viewers/testers of the prototype, 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.

Add an about page

1. First, let's add the [Flatpages App](#), which will allow us to add basic pages with HTML content.

Add flatpages to the `INSTALLED_APPS` section in `setings.py`:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'django.contrib.admin',  
9     'main',  
10    'django.contrib.flatpages',  
11 )
```

Update the `DATABASES` section in `settings.py`:

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         'NAME': 'test.db'  
5     }  
6 }
```

Update the `MIDDLEWARE_CLASSES` of `settings.py`:

```
1 MIDDLEWARE_CLASSES = (  
2     'django.middleware.common.CommonMiddleware',  
3     'django.contrib.sessions.middleware.SessionMiddleware',  
4     'django.middleware.csrf.CsrfViewMiddleware',  
5     'django.contrib.auth.middleware.AuthenticationMiddleware',  
6     'django.contrib.messages.middleware.MessageMiddleware',  
7     'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',  
8 )
```

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. Fi-

nally, go through the third course, *Advanced Web Development with Django*, for a very detailed explanation specific to Django, which will tie everything together.

Add the following pattern to *urls.py*:

```
1 url(r'^pages/', include('django.contrib.flatpages.urls')),
```

Add a new template folder within the “templates” directory called “flatpages”. Then add a default template by creating a new file, *default.html*, within that new directory. Add the following code to the file:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4 {{ flatpage.content }}
5 {% endblock %}
```

2. Sync the database and create a superuser:

```
1 $ python manage.py syncdb
```

NOTE: The *syncdb* command searches your *models.py* files and creates database tables for them. This is equivalent to the *migrate* command in Django 1.1/1.8.

You can use any username, email, and password that you’d like. For example:

- Username: admin
- Email: ad@min.com
- Password: admin

3. Update the About url in the *base.html* file:

```
1 <li><a href="/pages/about">About</a></li>
```

4. Launch the server and navigate to <http://localhost:8000/admin/>, login with the username and password you just created for the superuser, and then add the following page within the Flatpages section:

The HTML within the content portion:

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 too, but I highly recommend it. Cheers</p> <h3>Ready? Contact us!</h3></pre>

Figure 30.2: Django Flat Pages App - adding a new page

```
1 <br>
2
3 <p>You can add some text about yourself here. Then when you are
  done, just add a closing HTML paragraph tag.</p>
4
5 <ul>
6   <li>Bullet Point # 1</li>
7   <li>Bullet Point # 2</li>
8   <li>Bullet Point # 3</li>
9   <li>Bullet Point # 4</li>
10 </ul>
11 <br/>
12
13 <p><em>You can add a link or an email address <a
    href="http://www.realpython.com">here</a> if you want. Again,
    you don't have to, but I <strong>highly</strong> recommend it.
    Cheers! </em></p>
14
15 <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:

```
1 .
2  django_ecommerce
3      __init__.py
4      settings.py
5      urls.py
6      wsgi.py
7  main
8      __init__.py
9      models.py
10     tests.py
11     views.py
12  manage.py
13  templates
14     base.html
15     flatpages
16         default.html
17     index.html
18  test.db
```

Contact App

1. Create a new app:

```
1 $ python manage.py startapp contact
```

2. Add the new app to your *settings.py* file:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'django.contrib.admin',  
9     'main',  
10    'django.contrib.flatpages',  
11    'contact',
```

3. Setup a new model to create a database table:

```
1 from django.db import models  
2 import datetime  
3  
4  
5 class ContactForm(models.Model):  
6     name = models.CharField(max_length=150)  
7     email = models.EmailField(max_length=250)  
8     topic = models.CharField(max_length=200)  
9     message = models.CharField(max_length=1000)  
10    timestamp = models.DateTimeField(  
11        auto_now_add=True, default=datetime.datetime.now  
12    )  
13  
14    def __unicode__(self):  
15        return self.email  
16  
17    class Meta:  
18        ordering = ['-timestamp']
```

Does this make sense to you? The timestamp and the Meta() class may be new. If so, take a look at [this](#) link and [this](#) link from the Django documentation. In the latter case, we are simply deviating from the default ordering and defining our own based on the data added.

4. Sync the database:

```
1 $ python manage.py syncdb
```

5. Add a view:

```
1 from django.http import HttpResponseRedirect, HttpResponseRedirect
2 from django.template import RequestContext, loader
3 from .forms import ContactView
4 from django.contrib import messages
5
6
7 def contact(request):
8     if request.method == 'POST':
9         form = ContactView(request.POST)
10        if form.is_valid():
11            our_form = form.save(commit=False)
12            our_form.save()
13            messages.add_message(
14                request, messages.INFO, 'Your message has been
15                sent. Thank you.'
16            )
17            return HttpResponseRedirect('/')
18        else:
19            form = ContactView()
20            t = loader.get_template('contact.html')
21            c = RequestContext(request, {'form': form, })
22            return HttpResponseRedirect(t.render(c))
```

Here if the request is a POST - e.g., if the form is submitted - 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: You will learn *a lot* more about how forms work, and how to properly test them, in the third course. Check it out!

6. Update your Project's `urls.py` with the following pattern:

```
1 url(r'^contact/', 'contact.views.contact', name='contact'),
```

7. Create a new file within the “contacts” directory called *forms.py*:

```
1 from django.forms import ModelForm
2 from .models import ContactForm
3 from django import forms
4
5
6 class ContactView(ModelForm):
7     message = forms.CharField(widget=forms.Textarea)
8
9     class Meta:
10         model = ContactForm
```

8. Add another new file called *admin.py*:

```
1 from django.contrib import admin
2 from .models import ContactForm
3
4
5 class ContactFormAdmin(admin.ModelAdmin):
6     class Meta:
7         model = ContactForm
8
9 admin.site.register(ContactForm, ContactFormAdmin)
```

9. Add a new file to the templates directory called *contact.html*:

```
1 {% extends 'base.html' %}
2
3
4 {% block content %}
5
6 <h3><center>Contact Us</center></h3>
7
8 <form action='.' method='POST'>
9     {% csrf_token %}
10     {{ form.as_table }}
11     <br>
12     <button type='submit' class='btn btn-primary'>Submit</button>
```

```

13 </form>
14
15 {% endblock %}

```

See anything new? How about the `{% csrf_token %}` and `{{ form.as_table }}` 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_table` method simply renders the form as a table in your HTML. Check out some of the other ways you can render forms [here](#).

10. Update the Contact url in the *base.html* file:

```

1 <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 - model, view, template - fits into the whole.

What questions do you have? Add them to the support forum.

Did you notice the Flash message? See if you can recognize the code for it in the *views.py* file. You can read more about the messages framework [here](#), which provides support for Flask-like flash messages.

Updated structure:

```

1 .
2  contact
3      __init__.py
4      admin.py
5      forms.py
6      models.py
7      tests.py
8      views.py
9  django_ecommerce
10     __init__.py
11     settings.py

```

```
12     urls.py
13     wsgi.py
14 main
15     __init__.py
16     models.py
17     tests.py
18     views.py
19 manage.py
20 templates
21     base.html
22     contact.html
23     flatpages
24         default.html
25     index.html
26 test.db
```

User Registration with Stripe

In this last section, we will look at how to implement a payment and user registration/authentication system. For this payment system, registration is tied to payments. In other words, 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.

1. Update the path to the `STATICFILES_DIRS` in the `settings.py` file and add the “static” directory to the “project_name” directory:

```
1 STATICFILES_DIRS = (os.path.join(SITE_ROOT, 'static'),)
```

2. Create a new app:

```
1 $ python manage.py startapp payments
```

3. Add the new app to `settings.py`:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'django.contrib.admin',  
9     'main',  
10    'django.contrib.flatpages',  
11    'contact',  
12    'payments',
```

4. Install stripe:

```
1 $ pip install stripe==1.9.2
```

5. Update `models.py`:

```

1 from django.db import models
2 from django.contrib.auth.models import AbstractBaseUser
3
4
5 class User(AbstractBaseUser):
6     name = models.CharField(max_length=255)
7     email = models.CharField(max_length=255, unique=True)
8     # password field defined in base class
9     last_4_digits = models.CharField(max_length=4, blank=True,
10                                     null=True)
11     stripe_id = models.CharField(max_length=255)
12     created_at = models.DateTimeField(auto_now_add=True)
13     updated_at = models.DateTimeField(auto_now=True)
14
15     USERNAME_FIELD = 'email'
16
17     def __str__(self):
18         return self.email

```

SEE ALSO: This model replaces the default Django User model. For more on this, please visit the official [Django docs](#)

6. Sync the database:

```

1 $ python manage.py syncdb

```

7. Add the following import and patterns to *urls.py*:

```

1 from payments import views
2
3 # user registration/authentication
4 url(r'^sign_in$', views.sign_in, name='sign_in'),
5 url(r'^sign_out$', views.sign_out, name='sign_out'),
6 url(r'^register$', views.register, name='register'),
7 url(r'^edit$', views.edit, name='edit'),

```

8. Add the remaining templates and all of the static files from the files in the [repository](#). Take a look at the templates. These will make more sense after you see the app in action.

- 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*

9. Add a new file called *admin.py*:

```
1 from django.contrib import admin
2 from .models import User
3
4
5 class UserAdmin(admin.ModelAdmin):
6     class Meta:
7         model = User
8
9 admin.site.register(User, UserAdmin)
```

10. Add another new file called *forms.py*:

```
1 from django import forms
2 from django.core.exceptions import NON_FIELD_ERRORS
3
4
5 class PaymentForm(forms.Form):
6     def addError(self, message):
7         self._errors[NON_FIELD_ERRORS] = self.error_class([message])
8
9
10 class SigninForm(PaymentForm):
11     email = forms.EmailField(required=True)
12     password = forms.CharField(
13         required=True,
14         widget=forms.PasswordInput(render_value=False)
15     )
16
17 class CardForm(PaymentForm):
18     last_4_digits = forms.CharField(
19         required=True,
20         min_length=4,
21         max_length=4,
22         widget=forms.HiddenInput()
23     )
24     stripe_token = forms.CharField(required=True,
25                                     widget=forms.HiddenInput())
```

```

25
26
27 class UserForm(CardForm):
28     name = forms.CharField(required=True)
29     email = forms.EmailField(required=True)
30     password = forms.CharField(
31         required=True,
32         label=(u'Password'),
33         widget=forms.PasswordInput(render_value=False)
34     )
35     ver_password = forms.CharField(
36         required=True,
37         label=(u' Verify Password'),
38         widget=forms.PasswordInput(render_value=False)
39     )
40
41     def clean(self):
42         cleaned_data = self.cleaned_data
43         password = cleaned_data.get('password')
44         ver_password = cleaned_data.get('ver_password')
45         if password != ver_password:
46             raise forms.ValidationError('Passwords do not match')
47         return cleaned_data

```

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 forms align to the templates.

11. Update *payments/views.py*:

```

1 from django.db import IntegrityError
2 from django.http import HttpResponseRedirect
3 from django.shortcuts import render_to_response
4 from django.template import RequestContext
5 from payments.forms import SigninForm, CardForm, UserForm
6 from payments.models import User
7 import django_ecommerce.settings as settings
8 import stripe
9 import datetime
10
11 stripe.api_key = settings.STRIPE_SECRET
12

```

```

13
14 def soon():
15     soon = datetime.date.today() + datetime.timedelta(days=30)
16     return {'month': soon.month, 'year': soon.year}
17
18
19 def sign_in(request):
20     user = None
21     if request.method == 'POST':
22         form = SigninForm(request.POST)
23         if form.is_valid():
24             results =
25                 User.objects.filter(email=form.cleaned_data['email'])
26             if len(results) == 1:
27                 if
28                     results[0].check_password(form.cleaned_data['password']):
29                     request.session['user'] = results[0].pk
30                     return HttpResponseRedirect('/')
31             else:
32                 form.addError('Incorrect email address or
33                     password')
34         else:
35             form.addError('Incorrect email address or password')
36     else:
37         form = SigninForm()
38
39     print form.non_field_errors()
40
41     return render_to_response(
42         'sign_in.html',
43         {
44             'form': form,
45             'user': user
46         },
47         context_instance=RequestContext(request)
48     )
49
50 def sign_out(request):
51     del request.session['user']

```

```

50     return HttpResponseRedirect('/')
51
52
53 def register(request):
54     user = None
55     if request.method == 'POST':
56         form = UserForm(request.POST)
57         if form.is_valid():
58
59             #update based on your billing method (subscription vs
60             #one time)
61             customer = stripe.Customer.create(
62                 email=form.cleaned_data['email'],
63                 description=form.cleaned_data['name'],
64                 card=form.cleaned_data['stripe_token'],
65                 plan="gold",
66             )
67             # customer = stripe.Charge.create(
68             #     description = form.cleaned_data['email'],
69             #     card = form.cleaned_data['stripe_token'],
70             #     amount="5000",
71             #     currency="usd"
72             # )
73
74             user = User(
75                 name=form.cleaned_data['name'],
76                 email=form.cleaned_data['email'],
77                 last_4_digits=form.cleaned_data['last_4_digits'],
78                 stripe_id=customer.id,
79             )
80
81             #ensure encrypted password
82             user.set_password(form.cleaned_data['password'])
83
84             try:
85                 user.save()
86             except IntegrityError:
87                 form.addError(user.email + ' is already a member')
88             else:
89                 request.session['user'] = user.pk

```

```

89         return HttpResponseRedirect('/')
90
91     else:
92         form = UserForm()
93
94     return render_to_response(
95         'register.html',
96         {
97             'form': form,
98             'months': range(1, 12),
99             'publishable': settings.STRIPE_PUBLISHABLE,
100             'soon': soon(),
101             'user': user,
102             'years': range(2011, 2036),
103         },
104         context_instance=RequestContext(request)
105     )
106
107
108 def edit(request):
109     uid = request.session.get('user')
110
111     if uid is None:
112         return HttpResponseRedirect('/')
113
114     user = User.objects.get(pk=uid)
115
116     if request.method == 'POST':
117         form = CardForm(request.POST)
118         if form.is_valid():
119
120             customer = stripe.Customer.retrieve(user.stripe_id)
121             customer.card = form.cleaned_data['stripe_token']
122             customer.save()
123
124             user.last_4_digits = form.cleaned_data['last_4_digits']
125             user.stripe_id = customer.id
126             user.save()
127
128     return HttpResponseRedirect('/')

```

```

129
130     else:
131         form = CardForm()
132
133     return render_to_response(
134         'edit.html',
135         {
136             'form': form,
137             'publishable': settings.STRIPE_PUBLISHABLE,
138             'soon': soon(),
139             'months': range(1, 12),
140             'years': range(2011, 2036)
141         },
142         context_instance=RequestContext(request)
143     )

```

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.

I can assure you that if you are not utilizing the latter method then you are not learning. Ask in the support forum for help on this.

You can charge users either a one time charge or a recurring charge. This script is setup for the latter. To change to a one time charge, simply make the following changes to the file:

```

1  #update based on your billing method (subscription vs one time)
2  #customer = stripe.Customer.create(
3  #email = form.cleaned_data['email'],
4  #description = form.cleaned_data['name'],
5  #card = form.cleaned_data['stripe_token'],
6  #plan="gold",
7  #)
8  customer = stripe.Charge.create(
9  description = form.cleaned_data['email'],
10 card = form.cleaned_data['stripe_token'],
11 amount="5000",
12 currency="usd"
13 )

```

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 look like this:

```
1  .
2  contact
3      __init__.py
4      admin.py
5      forms.py
6      models.py
7      tests.py
8      views.py
9  django_ecommerce
10     __init__.py
11     settings.py
12     urls.py
13     wsgi.py
14  main
15     __init__.py
16     models.py
17     tests.py
18     views.py
19  manage.py
20  payments
21     __init__.py
22     admin.py
23     forms.py
24     models.py
25     tests.py
26     views.py
27  templates
28     base.html
29     cardform.html
30     contact.html
31     edit.html
32     errors.html
33     field.html
34     flatpages
35         default.html
36     home.html
```

```

37     index.html
38     register.html
39     sign_in.html
40     test.html
41     user.html
42 test.db

```

12. Update *main/views.py*:

```

1 from django.shortcuts import render_to_response
2 from payments.models import User
3
4
5 def index(request):
6     uid = request.session.get('user')
7     if uid is None:
8         return render_to_response('index.html')
9     else:
10        return render_to_response(
11            'user.html',
12            {'user': User.objects.get(pk=uid)}
13        )

```

13. Update the *base.html* template:

```

1 {% load static %}
2
3 <!DOCTYPE html>
4 <html lang="en">
5   <head>
6     <meta charset="utf-8">
7     <title>Your MVP</title>
8     <meta name="viewport" content="width=device-width,
9       initial-scale=1.0">
10    <meta name="description" content="">
11    <meta name="author" content="">
12
13    <!-- Le styles -->
14    <link
15      href="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap
16      rel="stylesheet">

```



```

14 <style>
15
16 body {
17     padding-top: 20px;
18     padding-bottom: 40px;
19 }
20
21 /* Custom container */
22 .container-narrow {
23     margin: 0 auto;
24     max-width: 700px;
25 }
26 .container-narrow > hr {
27     margin: 30px 0;
28 }
29
30 /* Main marketing message and sign up button */
31 .jumbotron {
32     margin: 60px 0;
33     text-align: center;
34 }
35 .jumbotron h1 {
36     font-size: 72px;
37     line-height: 1;
38 }
39 .jumbotron .btn {
40     font-size: 21px;
41     padding: 14px 24px;
42 }
43
44 /* Supporting marketing content */
45 .marketing {
46     margin: 60px 0;
47 }
48 .marketing p + h4 {
49     margin-top: 28px;
50 }
51
52 </style>
53 <script src="http://code.jquery.com/jquery-1.11.0.min.js"

```

```

54     type="text/javascript"></script>
55     {% block extrahead %}
56     {% endblock %}
57     <script type="text/javascript">
58     $(function(){
59         {% block jquery %}
60         {% endblock %}
61     });
62     </script>
63     <script src="https://js.stripe.com/v1/"
64         type="text/javascript"></script>
65     <script type="text/javascript">
66     //<![CDATA[
67     Stripe.publishableKey = '{{ publishable }}';
68     //]]>
69     </script>
70     <script src="{% get_static_prefix %}jquery.js"
71         type="text/javascript"></script>
72     <script src="{% get_static_prefix %}application.js"
73         type="text/javascript"></script>
74 </head>
75
76 <body>
77
78     <div class="container-narrow">
79
80         <div class="masthead">
81             <ul class="nav nav-pills pull-right">
82                 <li><a href="{% url 'home' %}">Home</a></li>
83                 <li><a href="/pages/about">About</a></li>
84                 <li><a href="{% url 'contact' %}">Contact</a></li>
85                 {% if user %}
86                 <li><a href="{% url 'sign_out' %}">Logout</a></li>
87                 {% else %}
88                 <li><a href="{% url 'sign_in' %}">Login</a></li>
89                 <li><a href="{% url 'register' %}">Register</a></li>
90                 {% endif %}
91             </ul>
92             <h3><span class="fui-settings-16 muted">Your
93                 MVP!</span></h3>

```

```

89     </div>
90
91     <hr>
92
93     {% if messages %}
94     <div class="alert alert-success">
95         <div class="messages">
96             {% for message in messages %}
97                 {{ message }}
98             {% endfor %}
99         </div>
100     </div>
101     {% endif %}
102
103     <div class="container-narrow">
104         {% block content %}
105         {% endblock %}
106     </div>
107
108     <hr>
109
110     <div class="footer">
111         <p>Copyright © 2014 <a href="http://www.yoursite.com">Your
            MVP</a>. Developed by <a
                href="http://realpython.com">Your Name Here</a>. Powered
                by Django.</P></p>
112     </div>
113
114 </div>
115
116 </body>
117 </html>

```

14. Update the large button on *index.html*:

```

1 <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.

1. Go to the Stripe test Dashboard at <https://dashboard.stripe.com/test/dashboard>.
2. The first time you go to this URL, it will ask you to login. Don't. Instead, click the link to sign up, then on the sign up page, click the link to skip this step. You should be taken to the account page.
3. Now, you should be able to access the keys, by going to this URL: <https://dashboard.stripe.com/account/apikeys>. If you have trouble with this step, comment in the support forum.
4. Take note of the keys, adding them to `settings.py`:

```
1 STRIPE_SECRET = 'sk_test_4QBquf6d5EzsnJC1fTI2GBGm'
2 STRIPE_PUBLISHABLE = 'pk_test_4QBqqGvCk9gaNn3pl1cwxcAS'
```

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 future date 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':

```
1 customer = stripe.Customer.create(
2     email = form.cleaned_data['email'],
3     description = form.cleaned_data['name'],
4     card = form.cleaned_data['stripe_token'],
5     plan="gold",
6 )
```

Let's setup 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:

- ID: gold

- Name: Amazing Gold Plan
- Amount: 20

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.

Finally, after you process the test/dummy payment, make sure the user is added to the database, using the SQLite database browser:

Back on Stripe, within the [dashboard](#) click *Payments*. You should see the \$20 charge. If you click on that charge, you can see the account details as well as the email address associated with that user.

Congrats! You now have a basic working application setup.

It's up to you to figure out what type of product or service you are offering the user, or you can check out the third course to see exactly how to build one.

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.

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!

Homework

1. Want a bonus challenge? The original app that accompanied this course broke. You can find the code for this within the *django_mvp* folder in the course exercises repo. Fix it.
2. Add additional functionality to this app. Think about what you'd like to implement. Then post your idea on the forum. Someone from Real Python will respond to steer you in the right direction.

Name

John Eskey

Email

jack@ryan.com

Password

Verify Password

Credit card number

42424242424242

Security code (CVC)

123

Expiration date

8

2013

Register

Figure 30.3: Django Ecommerce MVP Example 2 - registration

LIVE TEST

Q Search...

Test

GENERAL

Dashboard

Payments

Customers

Transfers

Recipients

SUBSCRIPTIONS

Plans

Coupons

REQUESTS

Events & Webhooks

Logs

jack@ryan.com

cus_2Hj3mQGHDU0Kdb

Customer Details

Update Customer Details

ID: cus_2Hj3mQGHDU0Kdb

Created: Jul 29, 2013

Email: jack@ryan.com

Description: John Eskey

Cards

Add Card

▶ Visa

****4242

8/2013

DEFAULT

Edit

Delete

Payments

Create Payment

\$20.00

— ch_2Hj3OYzujyZzXW

2013/07/29 05:54:18

▶

Subscription

Edit Subscription

Change Plan

End Subscription

Status: Active

Plan: Amazing Gold Plan (\$20.00/month)

Figure 30.4: Verify customer data within Stripe

638

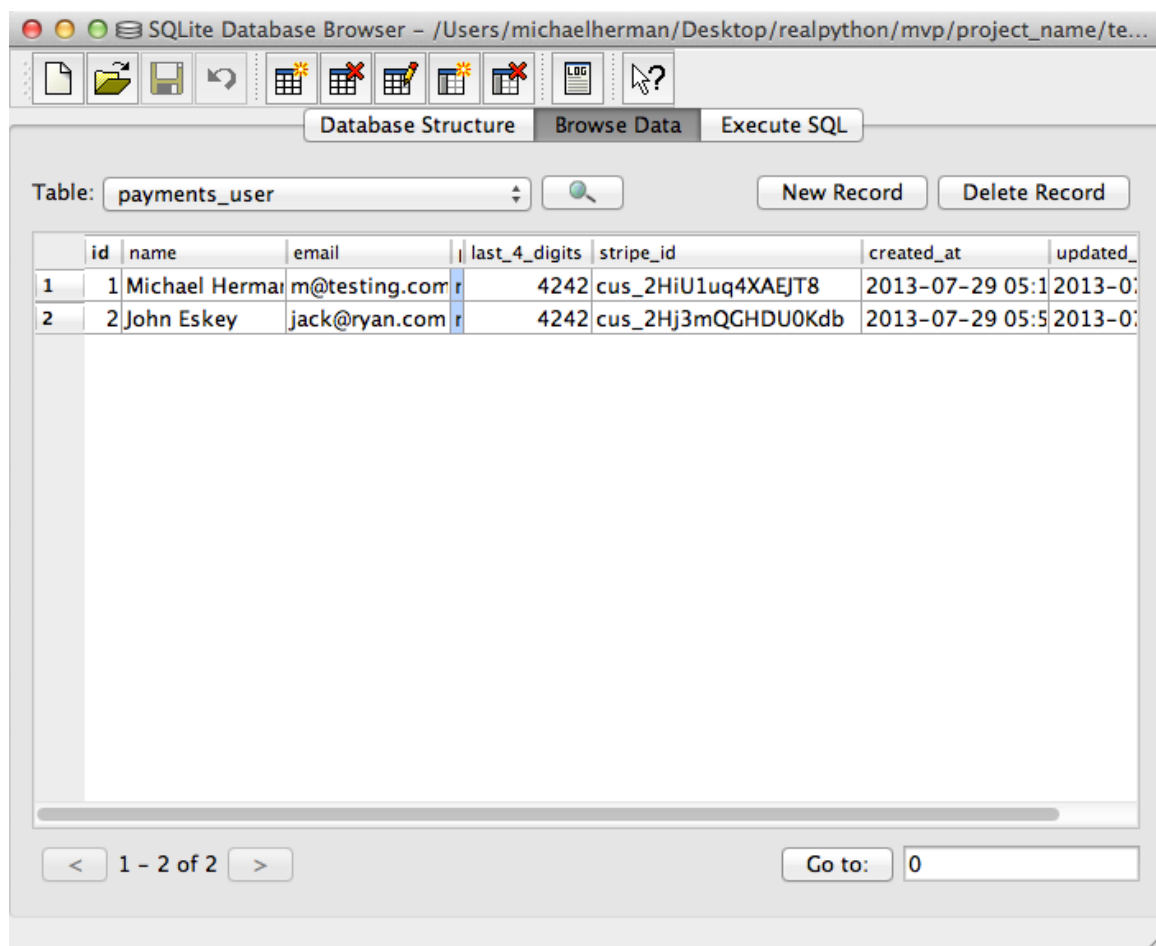


Figure 30.5: Verify customer data within SQLiteBrowse

Chapter 31

Appendix A: Installing Python

Let's get Python installed. Choose your Operating system.

Windows

Download

Start by downloading Python 2.7.7 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:\Python2.7. *You may need to be logged in as the administrator to run the install.*

NOTE: If you are working on course one you can stop here. However, if you plan on working through course two and/or three, please finish the full installation process.

Test

To test this install open your command prompt, which should open to the C:prompt, C:/>, then type:

```
1 \Python27\python.exe
```

And press enter. You should see:

```
1 Python 2.7.7 (r271:86832, ...) on win32
2 Type "help", "copyright", "credits" or "license" for more
  information.
3 >>>
```

Yay! You just started Python!

The >>> indicates that you are at the Python Shell (or prompt) where you can run Python code interactively.

To exit the Python prompt, type:

```
1 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 \Python27\python.exe.

Since you downloaded Python version 2.7.7, you need to add the following directories to your PATH:

- C:\Python27\
- C:\Python27\Scripts\
- C:\PYTHON27\DLLs\
- C:\PYTHON27\LIB\

Open your power shell and run the following statement:

```
1 [Environment]::SetEnvironmentVariable("Path",  
2     "$env:Path;C:\Python27\;C:\Python27\Scripts\;  
3     C:\PYTHON27\DLLs\;C:\PYTHON27\LIB\;", "User")
```

That's it.

To test to make sure Python is now on the PATH, open a *new* command prompt and then type python to load the Shell:

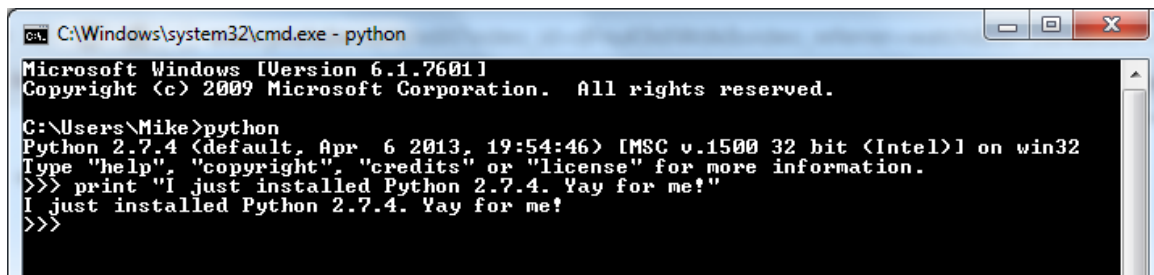


Figure 31.1: windows install

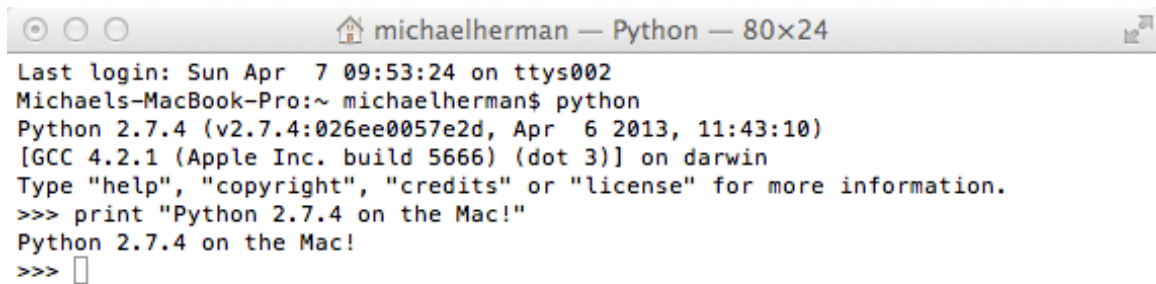
Congrats! You have Python installed and configured.

Video

Watch the video [here](#) for assistance. Note: Even though this is an older version of Python the steps are the same.

Mac OS X

All Mac OS X versions since 10.4 come with Python pre-installed. You can view the version by opening the terminal and typing `python` to enter the Shell. The output will look something like this:

A screenshot of a Mac OS X terminal window. The title bar at the top reads "michaelherman — Python — 80x24". The terminal text shows the last login time, the user's name and host, and the output of the 'python' command. The output indicates Python 2.7.4 is installed, showing its version string, GCC compiler version, and platform (darwin). It also provides instructions on how to get help, copyright, credits, or license information. A prompt is shown after the user enters a print statement.

```
Last login: Sun Apr  7 09:53:24 on ttys002
Michael's-MacBook-Pro:~ michaelherman$ python
Python 2.7.4 (v2.7.4:026ee0057e2d, Apr  6 2013, 11:43:10)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Python 2.7.4 on the Mac!"
Python 2.7.4 on the Mac!
>>> 
```

Figure 31.2: mac install

You need a Python version greater than 2.4. So, if you need to download a new version, download the latest [installer](#) for version 2.7.7.

Once downloaded, double-click the file to 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 2.7.7 (or whatever the latest version of Python is):

```
1 Python 2.7.7 (r271:86832, ...)
2 [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
3 Type "help", "copyright", "credits" or "license" for more
  information.
4 >>>
```

Congrats! You have Python installed and configured.

Linux

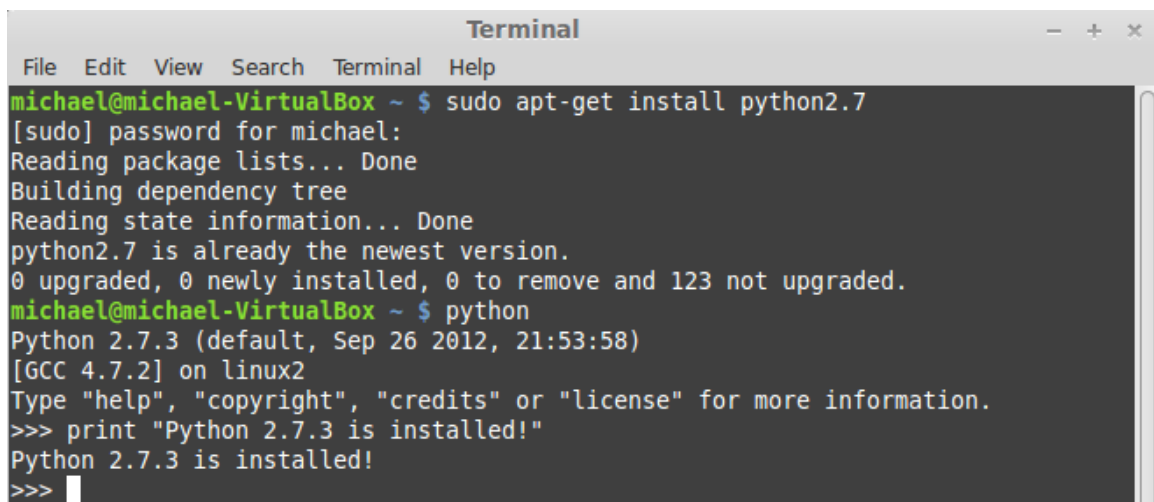
Most Linux distributions come pre-installed with the latest version of Python 2.7.x. If you have a version older than 2.4, please download the latest version. If you are using Ubuntu, Linux Mint, or another Debian-based system, enter the following command in your terminal to install Python:

```
1 sudo apt-get install python2.7
```

Or you can download the tarball directly from the official Python [website](#). Once downloaded, run the following commands:

```
1 $ tar -zxvf [mytarball.tar.gz]
2 $ ./configure
3 $ make
4 $ sudo make install
```

Once installed, fire up the terminal and type python to get to the shell:



```
Terminal
File Edit View Search Terminal Help
michael@michael-VirtualBox ~ $ sudo apt-get install python2.7
[sudo] password for michael:
Reading package lists... Done
Building dependency tree
Reading state information... Done
python2.7 is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 123 not upgraded.
michael@michael-VirtualBox ~ $ python
Python 2.7.3 (default, Sep 26 2012, 21:53:58)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Python 2.7.3 is installed!"
Python 2.7.3 is installed!
>>>
```

Figure 31.3: linux install

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.

Thanks!

Chapter 32

Appendix B: Supplementary Materials

Working with FTP

There are a number of different means of exchanging 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 be accessing an FTP server to view the main directory listing, upload a file, and then download a file.

Code:

```
1 import ftplib
2
3 server = ''
4 username = ''
5 password = ''
6
7 # Initialize and pass in FTP URL and login credentials (if
  applicable)
8 ftp = ftplib.FTP(host=server, user=username, passwd=password)
9
10 # Create a list to receive the data
11 data = []
12
```

```

13 # Append the directories and files to the list
14 ftp.dir(data.append)
15
16 # Close the connection
17 ftp.quit()
18
19 # Print out the directories and files, line by line
20 for l in data:
21     print(l)

```

Save the file. *DO NOT* run it just yet.

What's going on here?

So, we imported the `ftplib` library, which provides Python all 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:

```

1 server = 'ftp.debian.org'
2 username = 'anonymous'
3 password = 'anonymous'

```

Also add the following line just after you create the list:

```

1 # change into the debian directory
2 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:

```

1 -rw-rw-r-- 1 1176 1176 1066 Feb 15 09:23 README
2 -rw-rw-r-- 1 1176 1176 1290 Jun 26 2010
   README.CD-manufacture
3 -rw-rw-r-- 1 1176 1176 2598 Feb 15 09:23 README.html
4 -rw-r--r-- 1 1176 1176 176315 Mar 03 19:52
   README.mirrors.html

```



```

5 -rw-r--r--      1 1176      1176      87695 Mar 03 19:52
   README.mirrors.txt
6 drwxr-sr-x     15 1176      1176      4096 Feb 15 09:22 dists
7 drwxr-sr-x      4 1176      1176      4096 Mar 07 01:52 doc
8 drwxr-sr-x      3 1176      1176      4096 Aug 23 2013 indices
9 -rw-r--r--      1 1176      1176    8920680 Mar 07 03:15 ls-lR.gz
10 drwxr-sr-x      5 1176      1176      4096 Dec 19 2000 pool
11 drwxr-sr-x      4 1176      1176      4096 Nov 17 2008 project
12 drwxr-xr-x      3 1176      1176      4096 Oct 10 2012 tools

```

Next, let's take a look at how to download a file from a FTP server.

Code:

```

1 import ftplib
2 import sys
3
4 server = 'ftp.debian.org'
5 username = 'anonymous'
6 password = 'anonymous'
7
8 # Defines the name of the file for download
9 file_name = sys.argv[1]
10
11 # Initialize and pass in FTP URL and login credentials (if
   applicable)
12 ftp = ftplib.FTP(host=server, user=username, passwd=password)
13
14 ftp.cwd('debian')
15
16 # Create a local file with the same name as the remote file
17 with open(file_name, "wb") as f:
18
19     # Write the contents of the remote file to the local file
20     ftp.retrbinary("RETR " + file_name, f.write)
21
22 # Closes the connection
23 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:

```
1 $ python ftp_download.py README
```

Finally, let's take a look at how to upload a file to a FTP Server.

Code:

```
1 import ftplib
2 import sys
3
4 server = ''
5 username = ''
6 password = ''
7
8 # Defines the name of the file for upload
9 file_name = sys.argv[1]
10
11 # Initialize and pass in FTP URL and login credentials (if
    applicable)
12 ftp = ftplib.FTP(host=server, user=username, passwd=password)
13
14 # Opens the local file to upload
15 with open(file_name, "rb") as f:
16
17     # Write the contents of the local file to the remote file
18     ftp.storbinary("STOR " + file_name, f)
19
20 # Closes the connection
21 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.ofees.net>.

After you setup 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 any file found on your local directory.*

For example:

```
1 $ python 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

- See if you figure out how to navigate to a specific directory, upload a file, and then run a directory listing of that directory to see the newly uploaded file. Do this all in one script.

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.

Go ahead and install `pysftp`:

```
1 $ pip install pysftp
```

Now, let's try to list the directory contents from a remote server via SFTP.

Code:

```
1 import pysftp
2
3 server = ''
4 username = ''
5 password = ''
6
7 # Initialize and pass in SFTP URL and login credentials (if
  applicable)
8 sftp = pysftp.Connection(host=server, username=username,
  password=password)
9
10 # Get the directory and file listing
11 data = sftp.listdir()
12
13 # Closes the connection
14 sftp.close()
15
16 # Prints out the directories and files, line by line
17 for l in data:
18     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 setup 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.

Code:

```
1 import pysftp
2 import sys
3
4 server = ''
5 username = ''
6 password = ''
7
8 # Defines the name of the file for download
9 file_name = sys.argv[1]
10
11 # Initialize and pass in SFTP URL and login credentials (if
    applicable)
12 sftp = pysftp.Connection(host=server, username=username,
    password=password)
13
14 # Download the file from the remote server
15 sftp.get(file_name)
16
17 # Closes the connection
18 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 we outputted them to the screen.

For example:

```
1 $ python test_sftp_download.py remotefile.csv
```

Finally, let's take a look at how to upload a file to an SFTP Server.

Code:

```
1 # ex_appendixB.2c.py - SFTP File Upload
2
```

```

3 import pysftp
4 import sys
5
6 server = ''
7 username = ''
8 password = 'p@'
9
10 # Defines the name of the file for upload
11 file_name = sys.argv[1]
12
13 # Initialize and pass in SFTP URL and login credentials (if
    applicable)
14 sftp = pysftp.Connection(host=server, username=username,
    password=password)
15
16 # Upload the file to the remote server
17 sftp.put(file_name)
18
19 # Closes the connection
20 sftp.close()

```

When you run, make sure you specify a file name from your local directory on the command line.

For example:

```

1 $ python 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` module provided by Python is used to define the SMTP 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.

Code:

```
1 # Sending Email via SMTP (part 1)
2
3
4 import smtplib
5 from email.MIMEMultipart import MIMEMultipart
6 from email.MIMEText import MIMEText
7
8 # inputs for from, to, subject and body text
9 fromaddr = raw_input("Sender's email: ")
10 toaddr = raw_input('To: ')
11 sub = raw_input('Subject: ')
12 text = raw_input('Body: ')
13
14 # email account info from where we'll be sending the email from
15 smtp_host = 'smtp.mail.com'
16 smtp_port = '###'
17 user = 'username'
18 password = 'password'
19
20 # parts of the actual email
21 msg = MIMEMultipart()
22 msg['From'] = fromaddr
23 msg['To'] = toaddr
24 msg['Subject'] = sub
25 msg.attach(MIMEText(text))
26
27 # connect to the server
28 server = smtplib.SMTP()
29 server.connect(smtp_host, smtp_port)
```

```

30
31 # initiate communication with server
32 server.ehlo()
33
34 # use encryption
35 server.starttls()
36
37 # login to the server
38 server.login(user, password)
39
40 # send the email
41 server.sendmail(fromaddr, toaddr, msg.as_string())
42
43 # close the connection
44 server.quit()

```

Save. *DO NOT* run.

Since this code is pretty self-explanatory (follow along with the 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:

```

1 # email account info from where we'll be sending the email from
2 smtp_host = 'smtp.gmail.com'
3 smtp_port = 587
4 user = 'hermanmu@gmail.com'
5 password = "it's a secret - sorry"

```

I suggest using a Gmail account and sending and receiving from the same address at first to test it out, and then try sending 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:

```

1 # ex_appendixB.3b.py - Sending Email via SMTP (part 2)
2
3
4 import smtplib
5 from email.MIMEText import MIMEText
6 from email.MIMEMultipart import MIMEMultipart

```



```

7
8 def mail(fromaddr, toaddr, sub, text, smtp_host, smtp_port, user,
    password):
9
10     # parts of the actual email
11     msg = MIMEMultipart()
12     msg['From'] = fromaddr
13     msg['To'] = toaddr
14     msg['Subject'] = sub
15     msg.attach(MIMEText(text))
16
17     # connect to the server
18     server = smtplib.SMTP()
19     server.connect(smtp_host, smtp_port)
20
21     # initiate communication with server
22     server.ehlo()
23
24     # use encryption
25     server.starttls()
26
27     # login to the server
28     server.login(user, password)
29
30     # send the email
31     server.sendmail(fromaddr, toaddr, msg.as_string())
32
33     server.quit()
34
35 if __name__ == '__main__':
36     fromaddr = 'hermanmu@gmail.com'
37     toaddr = 'hermanmu@gmail.com'
38     subject = 'test'
39     body_text = 'hear me?'
40     smtp_host = 'smtp.gmail.com'
41     smtp_port = '587'
42     user = 'hermanmu@gmail.com'
43     password = "it's a secret - sorry"
44
45     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` module as part of the standard library which is used to define the IMAP client session implementation, used for accessing email. Essentially, we will be setting up our own mail server.

Code:

```
1 # Receiving Email via IMAPLIB
2
3
4 import imaplib
5
6 # email account info from where we'll be sending the email from
7 imap_host = 'imap.gmail.com'
8 imap_port = '993'
9 user = 'hermanmu@gmail.com'
10 password = "It's a secret - sorry!"
11
12 # login to the mail server
13 server = imaplib.IMAP4_SSL(imap_host, imap_port)
14 server.login(user, password)
15
16 # select the inbox
17 status, num = server.select('Inbox')
18
19 #fetch the email and the information you wish to display
20 status, data = server.fetch(num[0], '(BODY[TEXT])')
21
22 # print the results
23 print data[0][1]
24 server.close()
25 server.logout()
```

Notice how I used the same Gmail account as in 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.

If done correctly, you should see this:

```
1 Content-Type: text/plain; charset="us-ascii"
2 MIME-Version: 1.0
```

```
3 Content-Transfer-Encoding: 7bit
```

Test Python - What up!

“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 for loop to read the first 10 messages in your inbox.