

Django

IN ACTION

John Geewax

MEAP

 MANNING



©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=564>

Table of contents

Part 1 – Introducing Django

Chapter 1 – Using Django

Chapter 2 – Diving in: Django’s building blocks

Chapter 3 – Putting it into action: An internal messaging system

Part 2 – Advanced Django

Chapter 4 – URL patterns

Chapter 5 – Forms

Chapter 6 – Models

Chapter 7 – QuerySets

Chapter 8 – Views

Chapter 9 – Templates

Chapter 10 – Middleware

Chapter 11 – Users and sessions

Chapter 12 – Putting it into action: A social network web service

Part 3 – Django for businesses

Chapter 13 – Starting large projects

Chapter 14 – Maintaining large projects

Chapter 15 – Third-party libraries

Chapter 16 – Deployment

Appendix – A Python refresher for Django

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=564>

1

Meet Django

So you want to build a website, but not just any website; you want to build a dynamic website: a Web application. It's like Facebook combined with Amazon.com combined with Wikipedia and it runs in the cloud. You need a fully functional, mostly bug-free prototype up and running in about a month. Or maybe you just want to build a blogging engine, which you may or may not finish. Whichever the case, now's about the time you pull open your favorite text editor and start cranking away at a full custom WSGI stack in Python, right?

Maybe. There's definitely a time and a place for everything, but you obviously chose Python for a reason. Maybe it was for the dynamic typing, maybe for the readable syntax, probably not the threading, but there was an underlying reason: it's easier. You trade the performance gains of explicit memory management for automatic garbage collection because it's easier. It's easier because Python hides most of those pesky, nuanced, "gotcha" type problems so that you don't have to care about them.

This same principle applies in Web Development. You should be spending your time building your application, not general tools for all web applications. For most of your web app needs there is a good chance that someone else out there has stumbled across the exact same problem you're having, solved it, and published the solution for you to use, free of charge. Django is like a treasure trove of those little gems, and the best part is, it is being maintained constantly. Yes, even when you're asleep.

1.1 What is Django?

In a nutshell, Django is a big collection of useful tools that help you write more features with less code in less time. Sounds like there's a catch, right? Of course there is: Django, like all abstractions, is leaky and makes quite a few assumptions. These assumptions are true most of the time, but not all of the time, which means that sometimes they're going to end up biting you pretty hard. Luckily, when you get bitten it's usually because you're not doing things the way Django expects, and that means you just have to push your own opinions

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

aside and give “the Django way” a try. With that, let's take a quick tour of the tools that come with a brand new Django installation right out of the box.

1.1.2 Django's core toolset

Although it is debatable which components are considered to be in Django's core toolset, it is certainly at minimum fourfold, consisting of the URL dispatching layer, the ORM layer, the Templating layer, and the View layer. Each of these are discussed in a briefly, explaining more the purpose of each rather than how to use them. Don't worry though; you'll get plenty of chances to use each of these throughout the course of this book.

DIRECTING TRAFFIC WITH URL DISPATCHING

The URL dispatching layer allows you to steer away from scary looking URLs in favor of pretty, memorable ones. For example:

```
http://www.example.org/show_user.aspx?id=4  
might become  
http://www.example.org/users/4/
```

Not only is this easy on the eyes, but search engines might have trouble indexing your site with pages defined by their query string parameters. This isn't to say you can't make your URLs ugly; Django just gives you a hand in making it easy to be pretty.

LOADING AND PERSISTING DATA WITH DJANGO'S ORM

Django's ORM lets you search your database in Python rather than writing your own SQL queries. Gone are the days of tightly coupled querying in raw SQL such as in the PHP snippet in Listing 1.1.

Listing 1.1 Querying the database in PHP

```
<?php  
$host = "mysql-db.example.org";  
$user = "username";  
$password = "secret";  
$connection = mysql_connect($host, $user, $password);  
mysql_select_db("my_application", $connection);  
$sql = "SELECT `email` FROM `users` WHERE `username` = `jimmy`";  
$result = mysql_query($sql);  
$row = mysql_fetch_assoc($result);  
echo "Jimmy's e-mail address is: " . $row['email'];  
?>
```

In Django this becomes:

```
jimmy = User.objects.get(username='jimmy')  
print "Jimmy's e-mail address is: %s" % jimmy.email
```

Keep in mind that almost all of the queries you'll need to do can be done through Django's ORM, but if you find a situation where Django just cannot get the job done, you can always write your own SQL.

RENDERING DATA WITH DJANGO'S TEMPLATING ENGINE

Django's templating engine allows you to separate the layout and design from the data of your application. When it comes time to render a template, you're able to load up the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=564>

template you want, and pass in the data that should appear as variables or "context". This is useful for most web applications where you can render a template with specific data rather than building up a long string of HTML as a response. A quick example of a Django template is shown in Listing 1.2

Listing 1.2 A Sample Django Template

```
<html>
  <body>
    Welcome, {{ request.user.first_name }}!
  </body>
</html>
```

PUTTING THE PIECES TOGETHER WITH DJANGO'S VIEWS

To connect all the various components, Django maps URLs to view methods which are called for every request. Inside the views you can query the database, process and persist data, and then render the results to a template which then get pushed back to the browser. A simple view might look something like Listing 1.2.

Listing 1.2 A Sample View to Show a User Profile

```
from django.template import Template, Context
from django.http import HttpResponse

def show_user_profile(request, user_id):
    user = User.objects.get(id=id)

    template = Template("Username: {{ user.username }}")
    context = Context({'user': user})

    return HttpResponse(template.render(context))
```

#1 Views have Action Names

#2 Retrieves a User from the database

#3 Render a Template with this User

#4 Return the rendered template

As you can see from these examples, Django helps make your job as a developer easier by allowing you to spend more time focusing on the specifics of your application and less time on the more common problems of web development. It's almost as if "web development" is considered sort of a solved problem, and Django's dealing with that so that you can focus on your own problems. Like Virgil Cole from the movie Appaloosa, Django's "been doing this for a long time" and has a pretty impressive track record for handling web development problems.

1.1.3 A brief history of Django

Imagine you've just built your very first web application. It runs beautifully and there are only a few bugs--which you fix right away. Job well done. And then Mr. Jones comes around and assigns you your next project: another web app! You get right to work on Web App #2, and chances are you recognize a few of the same problems you saw in Web App #1. But you

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=564>

already solved those! Copy... Paste... and you're ready to go. Web App #2 is done in record time, Mr. Jones gives you a raise, and you go out to buy an iPhone and a Wii.

Then comes the request for Web App #3. About now you're probably putting Wii Tennis on hold and looking at the two applications wondering to yourself, "How are these two applications different?" You might sit and think for a while, click around in the source tree, notice a few mistakes you made in #1 that you avoided in #2, but once you find those pieces in common, you go ahead with the handy copy and

paste trick and put those common functions somewhere to be shared. Now you can do Web App #3 even faster. Not to mention a bug fix for Web App #2 now trickles into #1 and #3 and the soon-to-be-built Web App #4. Do this enough times and that pile of shared code gets pretty big, almost as if you've built some sort of "framework" of tools or something...

According to "The Definitive Guide to Django", this is exactly how Django started. Way back in 2003, in Lawrence, Kansas, a few developers started on the very same path described above and ended up with a framework of their own. However, unlike most "utils" directories, there was a pretty important difference: they kept working on the toolset. They continued fixing, adding, and redesigning the components until it was mature enough to graduate into the magical world of open source software. And it didn't stop there. Once in the wild the improvements continued--both on documentation and code. This means that, unlike a lot of open source code with no documentation, Django could be picked up and used by anyone willing to read a short tutorial. Soon roadmaps were written, bugs found, patches submitted, new versions released, conferences started, and new contributors added, and these things are all still ongoing without showing any signs of dying down.

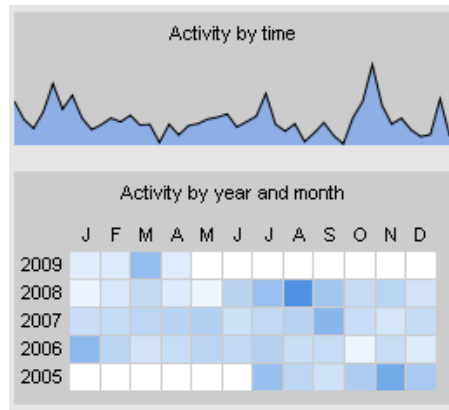


Figure 1 Django Development Activity

1.1.4 Why use Django?

We've already discussed the benefits of using a web framework for a building your next generation Web 2.0 application, so of all the frameworks available, why Django? What does it have over Ruby on Rails, or Turbo Gears, or Pylons, or the Zend Framework, or symfony? There are a myriad of programming languages and a huge variety of frameworks for each one, so why settle on this one called Django? All of these frameworks have their ups and downs, so with that in mind, maybe it would be better to discuss the reasons that a lot of developers end up with great results after choosing Django from the plethora of Web frameworks.

DJANGO'S DOCUMENTATION IS STELLAR

One extremely noticeable thing about Django's website (www.djangoproject.com) is the quality and quantity of the documentation available for all levels of experience with the framework. The getting started tutorial, although having some errors from time to time as new versions are released, is able to get an application up and running in record time. This means the chances of finding up-to-date documentation of the things discussed in this book are extremely high. It also means that when you are on a tight deadline to get a new feature released for Mr. Jones, you can find the solutions to your problems without staying up all night hacking with a 4-pack of Red Bull.

In addition to the official documentation, the user base of Django is friendly and very helpful. If you spend hours trying to solve a problem and cannot find any solutions in the documentation, a quick search on Stack Overflow brings up hundreds of questions asked and answered. The django-users Google Group is active and tickets on Django's Trac server are reviewed quickly, patches are posted often, and discussion is archived for the world to see and search. For many people, software is only as good as the support, and Django's marks here push it towards the front of the pack.

DJANGO HAS "JUST ENOUGH" MAGIC

One of Django's overarching design philosophies regards explicitness over implicitness. This means that Django's toolset will do most of the work for you while keeping the unexpected to the minimum. Take the following example:

```
jimmy = Person.objects.get(name="Jimmy")
```

Many developers would consider it strange if this line of code could, in any way, altered the database. For instance, if this line created a row for Jimmy in the case that he didn't already exist, that would end up in a book somewhere under the "gotchas" section. Luckily this isn't the case with Django, and if you'd like to create Jimmy if he doesn't exist, you can use `get_or_create`, like this:

```
jimmy, was_created = Person.objects.get_or_create(name="Jimmy")
```

The explicit functionality combined with the excellent documentation—which explains, for instance, that `get_or_create` returns a tuple to provide information on whether Jimmy was created or retrieved—makes development in Django more straight forward, without losing out on the usefulness of a reasonable amount of "Django magic".

DJANGO'S COMPONENTS PLAY NICE

Sometimes we have requirements we wish we didn't have. Mr. Jones might insist that you use Oracle for your database backend. Maybe your company spent millions of dollars developing a proprietary ORM in Python, and now you're stuck with it. Maybe you have an entire set of templates in written in Jinja or SimpleTAL and the hours to rewrite those for Django are simply not in the budget. Whatever the case, Django's components are in many ways "plug and play". Keep in mind that part of the appeal of Django is the entire package coming packaged together in one single framework, but sometimes you just can't get away with using Django right out of the box.

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

If Django were a “use it all or not at all” type of product, it would hardly be the ideal framework for any business with real deadlines and business requirements. Luckily, it happens to fit this bill rather well. Django’s internal components click together wonderfully, but using a different ORM like SQLAlchemy or a different templating engine like Mako is still relatively simple.

DJANGO'S ADMIN INTERFACE GETS YOU RUNNING IN NO TIME

Imagine you are building the world's biggest sandwich database. Right off the bat you want to start adding data for all the sandwiches you know. In order to do this, you'll have to build out an entire application that is able to accept input from a form in a browser, make sure it's all valid, and then store that information in your database. That doesn't sound too complicated, right?

Well, it does seem like an awful lot of work just to add "Peanut Butter & Jelly" to your newly created sandwich database. Under the covers you'll need to build a few templates, views, and forms all just to start the "data entry" portion of your project. For all that work, you might as well type up your data in Microsoft Excel, export it to CSV and then bulk load that with a Python script. That definitely doesn't sound like much fun. Luckily Django rushes to the rescue with its Admin Interface.

Django's Admin Interface is an incredibly useful tool that parses the definitions and constraints of your database from your model definitions and presents them to you as form fields in a full featured web UI. This tool, probably springing from Django's history in the fast paced world of newspapers, allows you to start adding and editing your data right away, with very little setup cost. Phew, no need for Excel.

Sure, Django might sound pretty great; being free, easy to learn, fast to get going, so what's the catch? I know it's considered poor form, but let's look this gift horse in the mouth and make sure everything's on the level.

1.2 A simple blog in 10 minutes

Now that you've gotten a peek under the hood of Django, let's take her for a test drive. This short little section will start you off by showing you how to design a bare-bones database and use Django's Admin Interface to view and modify your data. In the spirit of making use of Django's history as a content management oriented framework, let's make a quick and dirty blogging engine. The "Django blogging engine" is often considered a bit cliché for a first

application¹, but it's a very well understood concept, meaning we can focus on learning Django rather than the blog itself.

1.2.1 Getting set up

This section is written for a completely blank slate, so whether you're running a seasoned MacBook or your brand new Dell Mini 9" that just arrived in the mail, feel free to run through this portion like a tutorial and skip over anything you've already got set up. I'd venture to guess that most Python developers will skip a few of these, and if you're new to Python, you should start right here and get that installed.

INSTALLING PYTHON

The first thing you're going to need for the best blogging engine ever would be Python. In case you hadn't heard, Django was built in Python, so your blog will run on Python, meaning you'll probably need Python. Did I mention Python? Django will run fine on Python the 2.5 series of Python, but since the 2.6 series is backwards compatible, you should be able to use Python 2.6 without any problems.

If you're running Mac OS X, surprise! You've already got Python installed. Hop into Applications -> Utilities -> Terminal and run `python -V` and you should see a nice greeting telling you that Python 2.5.1 or 2.5.2 is installed. If you don't have the run time installed already, you can grab an Installer Disk Image (.dmg) from Python's download page (<http://python.org/download/>).

If you're running a flavor of Linux, you can `apt-get`, `yum`, or `emerge` Python, or grab the source tarball from Python's download page (see link above). Building Python from source is beyond the scope of this book, but if there are no tools to install Python on your platform chances are you've built a package from source before.

If you're running Windows, the Python downloads page (see link above) has an installer for both the 32- and 64-bit versions of Windows that you can install in a few clicks. When you're done, open up a Command Prompt (Start Menu -> Run -> type in `cmd`), and type `python -V`. You should get a nice response indicating that Python was installed. If you get an error about 'python' not being recognized as a command, it's more than likely your system path variable (where the command line looks for executables) isn't yet updated to look inside the newly installed Python directory, and restarting your computer should take care of this.

¹ In Cal Henderson's Keynote presentation for Django Con 2008, it seemed that not as many people were building this app as he'd thought! (<http://www.youtube.com/watch?v=i6Fr65PFqfk>).

INSTALLING EASY INSTALL

One of the great things about Python is the number and variety of packages and libraries² for you to use in all of your programs. However to make use of all of these packages, you'll need a convenient way to install them. Luckily, now that you have Python ready to go, you can grab `setuptools` from the Python Package Index (Pypi) which makes installing packages a breeze. Check out <http://pypi.python.org/pypi/setuptools#downloads> and download an installer (Windows) or egg (cross platform).

INSTALLING PYTHON DATABASE BINDINGS

Now that you have Easy Install set up and ready to go, you'll need to tell Python how to talk to your database of choice. Django comes with back ends to talk to most of the major relational databases, such as MySQL, PostgreSQL, Oracle, and even SQLite, but it relies on some Python libraries to do talking between SQL and Python. Choosing which database is best for your needs is a bit beyond the scope of this book, but you'll definitely want at least one of these--you can't use Django's Model layer without one! Since we'd like to keep this process as quick and painless as possible, let's just grab the SQLite bindings for Python. Although SQLite is far from ideal in a produce web environment, it's excellent for testing or small scale projects where you don't want the hassle and overhead of running a separate database server. Take a look at what happens when we use the `easy_install` command in Listing 1.3.

Listing 1.3 Sample Console Interaction to Easy Install SQLite

```
jjg@jjg-laptop:/$ sudo easy_install pysqlite
Searching for pysqlite
Reading http://pypi.python.org/simple/pysqlite/
Reading http://pysqlite.sourceforge.net/
Reading http://oss.itsystementwicklung.de/trac/pysqlite
Reading http://oss.itsystementwicklung.de/download/pysqlite/2.5/2.5.5/
Reading http://pysqlite.org/
Reading http://initd.org/pub/software/pysqlite/releases/2.3/2.3.3/
Reading http://initd.org/pub/software/pysqlite/releases/2.3/2.3.4/
Reading http://initd.org/tracker/pysqlite/wiki/PysqliteDownloads
Reading http://oss.itsystementwicklung.de/download/pysqlite/2.5/2.5.0/
Reading http://oss.itsystementwicklung.de/download/pysqlite/2.4/2.4.1/
Reading http://initd.org/pub/software/pysqlite/releases/2.4/2.4.0/
Reading http://initd.org/pub/software/pysqlite/releases/2.3/2.3.5/
Best match: pysqlite 2.5.5
Downloading
http://oss.itsystementwicklung.de/download/pysqlite/2.5/2.5.5/pysqlite-
```

² As a matter of fact, this is one of the major reasons the guys at Reddit decided to stick with Python. See <http://blip.tv/file/1951296> for a video of the Keynote speech at PyCon 2009.

```
2.5.5.tar.gz
Processing pysqlite-2.5.5.tar.gz
Running pysqlite-2.5.5/setup.py -q bdist_egg --dist-dir /tmp/easy_install-
_p_AT5/pysqlite-2.5.5/egg-dist-tmp-uoxzzx
zip_safe flag not set; analyzing archive contents...
Adding pysqlite 2.5.5 to easy-install.pth file
```

```
Installed /usr/lib/python2.5/site-packages/pysqlite-2.5.5-py2.5-linux-
i686.egg
Processing dependencies for pysqlite
Finished processing dependencies for pysqlite
```

That's it! Notice that Easy Install looked through all the various package repositories, found the best match, and then installed it correctly to our Python site-packages directory. Hopefully you've got the idea of Easy Install now and the rest of these prerequisite packages should be a breeze to get set up.

INSTALLING DJANGO

Finally we have our system ready to install Django. For the purposes of this book, Django 1.0 will work just fine, however some of the newer features, such as Query Set Aggregates, were added in the 1.1 alpha release, so it is probably best to get the latest stable version of Django from their website. Since Django has made a commitment to API stability from 1.0 onwards, all the code and examples in this book should be work perfectly with future 1.X versions of Django.

Jump on over to www.djangoproject.com/download/ and choose the latest official version. You can follow the instructions on that page (boils down to extracting the tarball and run `python setup.py install`) or you can use Easy Install by passing it the path to the tarball you've just downloaded: `easy_install Django-1.0.2-final.tar.gz` should work just fine. This will take you through a process similar to what you just saw when installing SQLite, but without all of the searching. This is because you've already told Easy Install which file is the "Best match."

STARTING YOUR PROJECT

Now that Django is installed, you're ready to create your first Django project. A Django project is really just a directory structure that Django creates for you, with some really handy utilities you'll need to test and develop your web application. To create this initial structure, Django comes with a tool called `django-admin.py`, located in your Python "Scripts" directory. To create a new project directory, simply run the script, passing the argument `createproject`:

```
jjg@jjg-laptop:~$ django-admin.py createproject project
This will create a project directory called "project" and the following files:
project/
  __init__.py
  manage.py
  settings.py
  urls.py
```

You might be wondering what these mysterious files actually do, so let's have a quick look at each.

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

The settings module holds exactly what you'd think: settings! In this file you can define what database backend you want to use, which applications you have installed, where your media files are located, or anything else relating to the configuration of your project. Later on you'll learn how to use Python's built in configuration file library to keep track of the settings for your project, but for now the "Django way" is to use this file.

The URLs module keeps track of what URLs map to which applications. If you had two applications, one for a messaging system and another for an online forum, you might set URLs starting with forum/ to point to the forum, and likewise for the messaging application.

The manage module is actually a script that helps you with the administrative aspects of your project. Want to create a new application? manage.py does that. How about turning on a development web server? manage.py does that too. What about synchronizing your model definitions to your database? manage.py. Need to export your data to JSON? manage.py for that too. As you can see manage.py does quite a bit of work and makes the administration of your project a breeze.

Typically, in addition to the basic structure Django builds, you'll need a place to store things like templates for the project or maybe CSS files you'll use throughout the project. Maybe the project will need more work down the line, and you'll need somewhere to keep all your documentation. It seems that Django tries not to assume too much, so you'll have to create those sub-directories on your own. A common full hierarchy might look something like this:

```
project/
  __init__.py
  manage.py
  settings.py
  urls.py
  docs/
    This might hold your documentation
  media/
    css/
    js/
    images/
  templates/
    This might hold the global project templates
```

Additionally, you should feel at home in this project folder. Go ahead and add anything here that you might need across your project.

THE DEVELOPMENT WEB SERVER

You now have a barebones Django project, but how do you know it actually works? To help make development easier, Django comes with a built in development web server, which you can use to test out your progress so far. This server is definitely not secure, so it's probably not a good choice to use in your production web site, but it can be extremely helpful during the development and debugging phase of your application. The development server will load up all of your project code (and reload it when you make changes) and let you see output from your application right in a terminal window. To turn on the development server, use the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=564>

"runserver" directive on the manage.py script in your project folder. The output from that command should look like Listing 1.4.

Listing 1.4 Sample Output from Starting the Django Development Server

```
jjg@jjg-laptop:~/project/$ python manage.py runserver
Validating models...
0 errors found

Django version 1.0-final-SVN-unknown, using settings 'my_project.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Looks like the server is running, but now what? Just open up a browser window and point it to the address the server is running on (in this example you can see its 127.0.0.1 on port 8000). You should see a very friendly screen looking something like Figure 2. Pretty exciting, right? Take a minute to enjoy that exciting feeling, and then let's get back to building.

1.2.2 Building the blog

In case you didn't notice from the helpful "It worked!" screen shown in Figure 2, Django is suggesting that you set some database settings and then start our first application. First, open up settings.py in your favorite Python editor. You should notice a few lines toward the top of the file that look like Listing 1.5.

Listing 1.5 Excerpt from Django's Settings File

```
DATABASE_ENGINE = '' # 'postgresql_psycopg2', 'postgresql', 'mysql',
'sqlite3' or 'oracle'.
DATABASE_NAME = '' # Or path to database file if using sqlite3.
DATABASE_USER = '' # Not used with sqlite3.
DATABASE_PASSWORD = '' # Not used with sqlite3.
DATABASE_HOST = '' # Set to empty string for localhost. Not used with
sqlite3.
DATABASE_PORT = '' # Set to empty string for default. Not used with
sqlite3.
```

Remember back to when you installed SQLite because it was "a simple database tool"? Here's where you actually get to use that. Set the DATABASE_ENGINE variable to 'sqlite3' as the comment suggests, and the DATABASE_NAME variable to 'project.db'. The other fields aren't needed for SQLite so Django won't even look at them. Setting these variables effectively tells Django to use SQLite to create a database file in your project directory called 'project.db'. Once you have that ready, you can move on to starting a new application for your project.

CREATING AN APPLICATION

A Django Application is a sub-component of a project that generally does one particular thing. For example, if Yahoo! were a Django project, Yahoo! Answers might be one app, Yahoo! mail might be another app, and Yahoo! Search might be another app. Of course, some applications might depend on others--to post a question on Yahoo! Answers you need a login which you get from Yahoo! Mail--and Django allows you to link across applications.

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

For example, you can use the `django.contrib.auth` application in your project to manage users across your project.

Now that you know what an application is, you'll need a way to create yours. You certainly could just create a directory and the appropriate files, but that seems obnoxious to do for every application you build. Django has a little tool that takes care of this for you right in `manage.py`. This is called 'startapp' and looks like this:

```
jjg@jjg-laptop:~/project/$ python manage.py startapp blog
```

If you look inside the 'project' directory, you'll see a new subdirectory was created called 'blog' which will hold your blog specific application code. The layout should look something like this:

```
project/  
  ...  
  blog/  
    __init__.py  
    models.py  
    views.py
```

This layout gives you the scaffolding of your new application, where you can build views and models that make up the core logic and data schema of your blog.

DEFINING MODELS

As you saw in the first section, Django can do some amazing things to help make accessing the database a painless process. In order to help Django along though, you'll need to define what the database should look like, that is, what database columns you need and what types they should be. You'll also give Django a bit of extra information that doesn't go directly to the database, but will help the framework validate that data being put into the database meets a certain standard of correctness.

For this blog it might be best to keep it simple, storing just one model, which we'll call "BlogEntry", that holds a few details about a single posting:

- 1) The title of the post,
- 2) The content of the post, and
- 3) When the post was written

For these, we'll use Django's `CharField`, `TextField`, and `DateField` objects³. If you open up `blog/models.py`, you'll notice that Django has a helpful prompt that you should "Create your models here". Do just that by adding the lines from Listing 1.6 to your `models.py` file.

Listing 1.6 Defining a Blog Entry (`blog/models.py`)

³ For a full reference of the available fields, a great place to look is Django's online documentation. The field reference is available here: <http://docs.djangoproject.com/en/dev/ref/models/fields/#ref-models-fields>.

```
class BlogEntry(models.Model):
    title = models.CharField()
    content = models.TextField()
    date_written = models.DateField()
```

That looks pretty readable, right? The next question you might ask is, "How do I know if it worked?" The answer lies back in `manage.py`, where we can synchronize our projects model definitions to the SQLite database that we configured beforehand. To do this, pass the `'syncdb'` directive to `manage.py` which, as you saw earlier, is in your projects root directory. The output from this should look like Listing 1.7.

Listing 1.7 Sample Output from `syncdb`

```
jjg@jjg-laptop:~/project/$ python manage.py syncdb
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
```

You just installed Django's auth system, which means you don't have any superusers defined.

Would you like to create one now? (yes/no): yes

Username: admin

E-mail address: admin@admin.com

Password: #1

Password (again):

Superuser created successfully.

Installing index for auth.Permission model

Installing index for auth.Message model

#1 I used "admin" here as the password

Well, everything seemed to go just fine, it looks like you even have a brand new super-user account, but there doesn't seem to be any mention of the blog. There's stuff about "auth" and "session" but no "blog". Perhaps something is missing from our configuration?

This is a common mistake when just getting started, and the solution is a simple one: whenever you add a new application, you need to tell Django that you want to use that application as part of your project. To do this, look for the last lines of `settings.py` which define a variable called `"INSTALLED_APPS"`. If you were to add `'project.blog'` to that list, it should install your Blog application.

```
jjg@jjg-laptop:~/project/$ python manage.py syncdb
Error: One or more models did not validate:
blog.blogentry: "title": CharFields require a "max_length" attribute.
```

Now we know that Django is at least looking at our `BlogEntry` model, but this isn't very encouraging. It might be helpful to know that Django's `CharField` is sort of the equivalent of MySQL's or PostgreSQL's "varchar" field, so you'd need to tell the database how long the field might potentially become. Next, if you were to check the documentation of `CharField` class, you'd see that Django doesn't just guess how long the field might become, and instead

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

requires that you explicitly provide this value. With that little bit of newfound knowledge, lets tweak the BlogEntry model slightly: a title shouldn't really be longer than 100 characters, so:

```
title = models.CharField(max_length=255)
```

Now, if you were to synchronize the database once again:

```
jjg@jjg-laptop:~/project/$ python manage.py syncdb
Creating table blog_blogentry
```

And that's all you need. You have set up a SQLite database, defined a few columns for storing a blog entry, and now you're ready to start adding and editing data with Django's Admin Interface!

1.2.3 Hooking up the Admin interface

It might seem a little scary that you've only added about five lines of code and are already moving on to editing data. Try not to worry too much, because as of now this would be a "write-only" blog. We haven't even though about how you show the world this blog since that would require writing a few new templates and views.

In order to get the Admin Interface connected to our project, we first need to set up some URL mapping to tell the web server that '/admin/' should connect to the Django Admin Interface. To do this, open up the urls.py file in your project directory, and take special note of the lines talking about un-commenting "to enable the admin".

Listing 1.7 Default Django URL Mappings

```
# Uncomment the next two lines to enable the admin:
# from django.contrib import admin          #1
# admin.autodiscover()                     #1
...
urlpatterns = patterns('',
    ...
    # Uncomment the next line to enable the admin:
    # (r'^admin/(.*)', admin.site.root),    #1
)
```

#1 Uncomment these lines as the prompts say

These lines are responsible for telling Django to send any requests for '/admin/' onto Django's Admin site root. Once you have that done, spin up the development server as you did before (with `python manage.py runserver`) and try navigating to <http://localhost:8080/admin/>. This should look something like Figure 3.

Whoops! It looks like something definitely went wrong here! Luckily this error message seems rather self explanatory: "Put 'django.contrib.admin' in your INSTALLED_APPS setting in order to use the admin application." Open up settings.py and do just that, and keep in mind that the order of those applications doesn't matter. Once you've turned back on the development web server, you should see a page looking something like Figure 4.

Try logging in with the username and password that you chose from before, and it should bring you further to a page looking like Figure 5.

Wow! That looks pretty great! This fully built interface and only a few lines of code written! But wait, once again it looks like the BlogEntry model is missing! There might be some extra configuration needed to tell the Admin app about your Blog.

That's exactly it, and this configuration goes in a file called `admin.py` inside the `blog` directory. This file will hold any special information pertaining to the Blog and its use in the Admin application. To tell the admin app about the blog, you'll need this file to look like Listing 1.8.

Listing 1.8 Registering the Blog Entry Model (`blog/admin.py`)

```
from project.blog.models import BlogEntry
from django.contrib import admin
admin.site.register(BlogEntry)
```

This little snippet tells the Admin app that you'd like to manage the BlogEntry model through the Admin Interface. Now, if you refresh the admin page, it should look something like Figure 6.

Inside this interface you can create new blog entries, edit existing entries, or delete old entries. It's pretty impressive that Django can give you a full CRUD (Create, Read, Update, and Delete) interface with only about four lines of code, but this certainly can't be your entire web application. Actually, it's more like half of the application: the write-only side. This next section will take you through building the other end of the application: showing the blog entries to the public.

1.3 Beyond the Admin interface: listing blog entries

The Admin Interface is perfectly usable for posting and modifying content, but chances are you'll want to do more than just post some blog entries that you never show to anyone else. Currently you have a write-only blog, which really isn't all that useful. In order to build out the reading end of the blog, you'll have to do a bit more work.

This might not make sense. How could reading from the database take more code than writing and editing in the Admin Interface? The simple answer is that Django does a lot of guesswork when pulling together the Admin Interface (notice in Figure 6 that the objects listed are called "BlogEntrys" rather than the more grammatically correct "Blog Entries"). When you're building public-facing pages, Django will do a better job of getting out of your way. Hopefully as you continue along with building this blog you'll understand why this is definitely a good thing, but for now you'll just have to trust that Django will still do most of the heavy lifting, you just need to tell it what to lift.

In this process of telling Django what to do, you'll get a chance to become more familiar with three of Django's components that weren't really touched on earlier: views, models (specifically with Django's ORM) and templates. Additionally you'll get some extra interaction with the URL Mapping component which you briefly fiddled with while connecting the wires for the Admin Interface. Now it's time to move onto building a page to list all your blog entries.

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

1.3.1 Rendering blog entries with Django templates

Although it may seem sort of backwards, a convenient place to begin this "list all blog entries" task is to design a Django template. For those of you familiar with templating engines, this should seem like a piece of cake with just some new syntax. If you've never touched a templating engine, don't worry, templates are really easy to learn and understand, and usually follow the structure of the programming language they're written in. Django's templates are no exception and follow pretty closely to plain old Python.

Since the main purpose of any templating engine is to separate the layout of data from the content, you'll need to think of Django's templates as the "layout" portion, and the data you'll later pass to that template as the "content" portion. This means that you can build the template to make use of different variables (called the "Context" in Django) all the while knowing that later the template and context will get mixed together to become the actual HTML that gets sent down to your browser.

With that in mind, the first step to building a Django template is creating the template file. A Django template is really nothing more than a simple HTML file that has Django's special markup littered throughout. Since it's really nothing more than HTML, you can create in it your favorite text editor as a plain text file. To keep track of the templates for your blog, now might be a good time to create a 'templates' directory inside the root project directory. Once that's done, you can create a directory for your "blog" templates and then a file called "list.html" and place it inside the blog directory you just created. Your directory structure should slightly resemble the following:

```
project/
...
blog/
... #1
templates/
  blog/ #2
    list.html
```

#1 Your application code goes here

#2 Your blog templates go here

VARIABLE SUBSTITUTION

So far there's been mention of template contexts but no discussion of how to actually use variables in a Django template. It's actually really simple: to tell Django to "find and replace" a variable with its value, just surround the variable name in curly braces like this: "<h1>{{ name }}</h1>". Additionally, Django respects the dot operator (.) on objects so that if you passed an object (say "person") who had a name, you can substitute the person's name like this: "<h1>{{ person.name }}</h1>". Although the spaces on either side of the variable name are not syntactically necessary, they are a stylistic convention strongly encouraged by most Django developers for clarity and consistency.

Now that you know how to substitute some variables, you can probably figure out how to display a single blog entry. You'll probably want to show all the details each entry, including the title, the contents, as well as the date it was posted. An example of a template which accepts a "blog_entry" context variable is shown in Listing 1.9.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=564>

Listing 1.9 Sample Template Rendering a Single Blog Entry

```
<html>
  <body>
    <h1>My First Blog</h1>
    <h2>{{ blog_entry.title }}</h2>
    <h5>Posted on {{ blog_entry.date_written }}</h5>
    <p>{{ blog.content }}</p>
  </body>
</html>
```

This template is not too stylish, as CSS is a bit outside the scope of this section, but it will correctly display a single blog entry's details in an HTML page once rendered by Django. That wasn't too difficult, right?

ITERATION

Although this variable substitution looks fine for a single blog entry, it would be a real pain if you had to pass "blog_entry1", "blog_entry2", "blog_entry3", ..., "blog_entry50" for each entry you wanted to display. It would also be scary if you were forced to copy-and-paste the construct to display each entry over and over for each entry. Fortunately this isn't necessary as Django templates have a construct, called a "template tag", which works almost identically to Python's "for" construct.

To use Django's template tags, rather than using two curly braces, use a curly brace and a percent sign like this: "{% ... %}". Inside the construct you can use almost the exact same code that you'd use in Python, as show in Listing 1.10.

Listing 1.10 Sample Template Rendering List of Blog Entries

```
<html>
  <body>
    <h1>My First Blog</h1>
    {% for blog_entry in blog_entries %}                                #1
      <h2>{{ blog_entry.title }}</h2>
      <h5>Posted on {{ blog_entry.date_written }}</h5>
      <p>{{ blog_entry.content }}</p>
    {% endfor %}                                                       #2
  </body>
</html>
```

#1 Note the similarity to Python's for-loop

#2 Notice that the loop must be explicitly closed

One small difference is that you'll need an end tag to signal when the block is finished. This is because Python's style of using indentation doesn't work as well in template markup with HTML, and although it may seem a bit odd at first, it becomes second nature over time. Notice that while inside the "for" block you have access to an extra context variable called "blog_entry" which represents the current item in the list being iterated over. Inside the block you also have access to all of the other context variables, but the "blog_entry" variable is special in that once outside this block it will disappear.

Although this short template also won't be very pretty, it will definitely accomplish the job of displaying all of the blog entries on the page. Go ahead and save Listing 1.10 in your

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

list.html template and you can then move on to learn how to actually render the template to HTML using Django.

1.3.2 Hooking the pieces together with views

With the template safely tucked away in your blog templates folder, you'll now need a way to connect blog entry data from your SQLite database to the template and then send the correctly rendered template down to the browser. This is exactly the purpose of Django's views.

DEFINING A LIST VIEW

To get started on your first view, take a look at the file called views.py in the blog application's directory (blog/views.py). This file is a bit bare, with a quasi-friendly prompt to "Create your views here," so it might be a bit unclear what you're supposed to do. Since you need a view that "lists all of the blog entries", you can create a simple function called "list_all" that takes as its sole argument the Django request (traditionally called "request")⁴. The barebones view might look something like Listing 1.11, which will simply return some text to the page.

Listing 1.11 A Barebones View Definition

```
from django.http import HttpResponse

def list_all(request):
    return HttpResponse("This should list all blog entries.")
```

Now that you've defined this specific view, you'll need to tell Django to do more than just send down some placeholder text. To pull in all of the blog entries, you'll first need to import the BlogEntry model and then you can use the BlogEntry model's "objects" attribute to retrieve all of the BlogEntry models as shown in Listing 1.12.

Listing 1.12 Retrieving all BlogEntry Instances

```
from django.http import HttpResponse
from project.blog.models import BlogEntry

def list_all(request):
    blog_entries = BlogEntry.objects.all()
    return HttpResponse("This should list all blog entries.")
```

There is a lot of work being done under the covers here, and the actual "magic" happening will be discussed in much more detail later on, so for now just know that this

⁴ Django adheres to a particular coding style (which includes the Python standard PEP 8) in order to keep the code base consistent and easy to understand and maintain, located here: <http://docs.djangoproject.com/en/dev/internals/contributing/#view-style>.

"`BlogEntry.objects.all()`" method returns an iterable object of `BlogEntry` model instances. This means that if you were to loop over the result of this method, each item in the series would be a single `BlogEntry` model instance with all the data populated from the database. This also means that if you were to pass the "blog_entries" variable to the template you built earlier, the for loop line, "`{% for blog_entry in blog_entries %}`", would work just fine.

Even though you've retrieved the data you need from your database, this view will still send down placeholder text. In order to return a rendered template you'll need to load a few other helper modules: one for locating the template you created earlier and one for creating the template's context. After that you can return the rendered template as a string instead of the placeholder text.

LOCATING THE TEMPLATE

Although you currently only have one single template, it's possible that throughout the lifetime of your project there will be far more than just one template. Obviously you could hard-code the path to the template in your view and use Python's `open` command to read in the contents, but that will probably end up limiting you down the line when you need to deploy your application on many different servers, with the templates living in different directories. Thankfully Django has a neat technique to prevent this potential headache.

For the sake of keeping all the templates organized, Django allows you to use different "template loaders", which are simply search functions that use different methods to do a lookup for a template by its name. Django comes with a few built-in template loaders, and also makes it easy for you to create your own custom template loaders, which you'll see later on. As an added bonus, plenty of custom template loaders have been created and shared online by other Django developers, and these are very easy to incorporate into your Django projects.

The most basic template loader, which is enabled by default, is very similar to how Linux or Windows searches through your `PATH` environment variable for executables when you type a command in the terminal. Therefore, in the same way that you must set up your `PATH` variable, this particular template loader requires you to specify which directories it should check when searching for your templates. This should be done in `settings.py` under the `TEMPLATE_DIRS` tuple as illustrated in Listing 1.13.

Listing 1.13 Sample `TEMPLATE_DIRS` Definition

```
TEMPLATE_DIRS = (  
    "/home/jjg/project/templates",      #1  
)
```

#1 These should be absolute rather than relative paths

With that configuration option in place, you'll need to tell Django to load your template. This is as simple as importing a function called "get_template" and passing it the name of the template you'd like to load. This will return a `Template` instance which can later be

rendered after you create a template context. Using the `get_template` method should make your `list_all` view look something like Listing 1.14.

Listing 1.14 Loading a Template in the List All View

```
from django.http import HttpResponse
from django.template.loader import get_template          #1
from project.blog.models import BlogEntry

def list_all(request):
    blog_entries = BlogEntry.objects.all()
    template = get_template("blog/list.html")           #2
    return HttpResponse("This should list all blog entries.")
#1 Import the template loader
#2 Load the template by name
```

Once that is in place, Django will know to look in your project's templates directory for a template called "blog/list.html".

CREATING THE TEMPLATE CONTEXT

Now that you have a hold of the template object, you'll need to build the context necessary to render it correctly with the data that you pulled from your database. Since template contexts are really just key-value pairs, it seems appropriate that Django would use a Python dictionary for handling this portion of the templating, and this is mostly what Django does but with a slight twist.

Django provides a `Context` object which is a lot like a dictionary, that is, you can use dictionary-like accessors with `__getitem__` and `__setitem__`, except it is made to be extended as you'll see in the more advanced sections. Since these context objects are so similar to dictionaries, creating one is as simple as importing the `Context` object and instantiating it from a dictionary of your context variables as you can see in Listing 1.15.

Listing 1.15 Instantiating a Context Object

```
from django.http import HttpResponse
from django.template.loader import get_template
from django.template import Context                    #1
from project.blog.models import BlogEntry

def list_all(request):
    blog_entries = BlogEntry.objects.all()
    template = get_template("blog/list.html")
    context = Context({'blog_entries': blog_entries})  #2
    return HttpResponse("This should list all blog entries.")
#1 Import the Context object
#2 The keys here are the variable names you can use in your template
```

RENDERING THE TEMPLATE

After all that setup is done, rendering the template couldn't be simpler. The `Template` instance returned by `get_template` has a helper method called "render". At the risk of restating the obvious, this method will take in the `Context` instance you created as its sole argument and return a string that is the template rendered appropriately. That is, the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=564>

content of this string is your fully rendered HTML page. This means that all you'll need to do to send down the fully rendered template to the browser is replace the placeholder text with `template.render(context)` as shown in Listing 1.16.

Listing 1.16 Returning the Rendered Template

```
from django.http import HttpResponse
from django.template.loader import get_template
from django.template import Context
from project.blog.models import BlogEntry

def list_all(request):
    blog_entries = BlogEntry.objects.all()
    template = get_template("blog/list.html")
    context = Context({'blog_entries': blog_entries})
    return HttpResponse(template.render(context)) #1

#1 Return the rendered template
```

Although you'll see later that there are shortcuts to make this process even quicker, this view should accurately retrieve, render, and return a list of all your blog entries to a web page. Now, just as with the Admin Interface, you'll need to connect the wires to tell your web server that a specific URL means "call the `list_all` view".

1.3.3 Flipping the switch: adding a URL pattern

Once you've completed building the view that decides what data to retrieve and which template to use to render that data, you'll need to give the web server a route that points a specific URL to this view. This could be any URL you want, but since the project is a blog and this is the only view, this will show you how to connect the `list_all` view to the website root.

To get started, let's revisit the URL mappings we initially tweaked to get the Admin Interface working, located in `project/urls.py`. You'll notice that the URL patterns are really a set of tuples with the first item being what looks like a regular expression. Although it isn't very clear, be sure to take special note that the initial forward slash is not needed on the regular expression path. This means that a match for the website root would simply be the empty regular expression: `^$`. This will match on the following URL when using the development server <http://127.0.0.1:8000/>.

The examples aren't exactly clear on what the second item in this list of tuples is, since it's not very obvious what `admin.site.root` is, or what is returned by the `include` function mentioned in the commented out examples. For now, it will suffice to know that the second argument can be a path to a Python module and view method. For example, since our method `list_all` is located in the module `project.blog.views`, a valid path would be `project.blog.views.list_all`. With that knowledge, you can add a new URL pattern by appending a second tuple after the mapping you added earlier for the Admin Interface. This should make your `urls.py` file look something like Listing 1.17.

Listing 1.17 Mapping the Website Root to the List All View

©Manning Publications Co. Please post comments or corrections to the Author Online forum: <http://www.manning-sandbox.com/forum.jspa?forumID=564>

```
urlpatterns = patterns('',
    ...
    # Uncomment the next line to enable the admin:
    (r'^admin/(.*)', admin.site.root),
    ...
    (r'^$', 'project.blog.views.list'),          #1
)
```

#1 Add your new URL mapping here

With that final piece of the puzzle in place, you can turn on the development web server just like you did when running the Admin Interface, and navigate to <http://127.0.0.1:8000/>. There you should see your fully rendered blog, which should look something like Figure 7. Notice that if you log into the Admin Interface, which is still located at <http://127.0.0.1:8000/admin/>, you can add or update blog entries and simply refresh to see the changes on the main page.

1.4 Summary

Hopefully after seeing how easy it is to pull together a barebones Django web application, you're in agreement that Django can make your life much easier than if you had to start all of this work from scratch. Think of all the HTTP internals that weren't even mentioned, or the database querying that you didn't have to write by hand, or the session management and user authentication that "just worked." In hindsight that stuff is pretty nice, right?

With that in mind, it's important to remember that although Django is an incredibly powerful piece of code, and definitely a good tool to have in your arsenal of skills, all this raw power can sometimes get out of control. This chapter walked you through using the basics of Django to get a project built quickly, but in many cases rapid development isn't always the ultimate goal. For example, in order to use Django in any sort of high-availability scenario it's simply vital to know how to break down the more advanced concepts into basic, almost primitive, ones. Rapid development is great, but when it comes time to optimize your application for speed, you'll need to be really confident in your understanding of the building blocks Django uses to simplify your web application.

With that in mind, this next chapter will discuss in more depth what is happening under the covers, hopefully providing a bit more insight on how the full Django stack works as a whole. Additionally the next chapter should help build up your intuition on knowing at least where to start to look if problems should arise in your application.