V Skills

Skills for a secure future

Certified Django Developer Sample Material

V-Skills Certifications

A Government of India
&
Government of NCT Delhi Initiative

V-Skills

Skills for a secure future

1. INTRODUCTION

This book is about Django, a Web development framework that saves you time and makes Web development a joy. Using Django, you can build and maintain high-quality Web applications with minimal fuss. At its best, Web development is an exciting, creative act; at its worst, it can be a repetitive, frustrating nuisance. Django lets you focus on the fun stuff — the crux of your Web application — while easing the pain of the repetitive bits. In doing so, it provides high-level abstractions of common Web development patterns, shortcuts for frequent programming tasks, and clear conventions for how to solve problems. At the same time, Django tries to stay out of your way, letting you work outside the scope of the framework as needed.

The goal of this book is to make you a Django expert. The focus is twofold. First, we explain, in depth, what Django does and how to build Web applications with it. Second, we discuss higher-level concepts where appropriate, answering the question "How can I apply these tools effectively in my own projects?" By reading this book, you'll learn the skills needed to develop powerful Web sites quickly, with code that is clean and easy to maintain.

1.1 What Is a Web Framework?

Django is a prominent member of a new generation of *Web frameworks*. So what exactly does that term mean?

To answer that question, let's consider the design of a Web application written using the Common Gateway Interface (CGI) standard, a popular way to write Web applications circa 1998. In those days, when you wrote a CGI application, you did everything yourself — the equivalent of baking a cake from scratch. For example, here's a simple CGI script, written in Python, that displays the ten most recently published books from a database:

```
#!/usr/bin/python
import MySQLdb
print "Content-Type: text/html"
print "<html><head><title>Books</title></head>"
print "<body>"
print "<h1>Books</h1>"
print "<ul>"
connection = MySQLdb.connect(user='me', passwd='letmein', db='my_db')
cursor = connection.cursor()
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC LIMIT 10")
for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]
print "</ul>"
print "</body></html>"
connection.close()
```

This code is straightforward. First, it prints a "Content-Type" line, followed by a blank line, as required by CGI. It prints some introductory HTML, connects to a database and executes a query that retrieves the latest ten books. Looping over those books, it generates an HTML unordered list. Finally, it prints the closing HTML and closes the database connection.

With a one-off dynamic page such as this one, the write-it-from-scratch approach isn't necessarily bad. For one thing, this code is simple to comprehend — even a novice developer can read these

16 lines of Python and understand all it does, from start to finish. There's nothing else to learn; no other code to read. It's also simple to deploy: just save this code in a file called latestbooks.cgi, upload that file to a Web server, and visit that page with a browser. But as a Web application grows beyond the trivial, this approach breaks down, and you face a number of problems:

- ✓ What happens when multiple pages need to connect to the database? Surely that database-connecting code shouldn't be duplicated in each individual CGI script, so the pragmatic thing to do would be to refactor it into a shared function.
- ✓ Should a developer *really* have to worry about printing the "Content-Type" line and remembering to close the database connection? This sort of boilerplate reduces programmer productivity and introduces opportunities for mistakes. These setup- and teardown-related tasks would best be handled by some common infrastructure.
- ✓ What happens when this code is reused in multiple environments, each with a separate database and password? At this point, some environment-specific configuration becomes essential.
- ✓ What happens when a Web designer who has no experience coding Python wishes to redesign the page? Ideally, the logic of the page — the retrieval of books from the database — would be separate from the HTML display of the page, so that a designer could edit the latter without affecting the former.

These problems are precisely what a Web framework intends to solve. A Web framework provides a programming infrastructure for your applications, so that you can focus on writing clean, maintainable code without having to reinvent the wheel. In a nutshell, that's what Django does.

1.2 MVC

Let's dive in with a quick example that demonstrates the difference between the previous approach and that undertaken using a Web framework. Here's how you might write the previous CGI code using Django:

```
# models.py (the database tables)
from django.db import models
class Book(models.Model):
    name = models.CharField(maxlength=50)
    pub_date = models.DateField()

# views.py (the business logic)
from django.shortcuts import render_to_response
from models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list':
book_list})

# urls.py (the URL configuration)
from django.conf.urls.defaults import *
import views

urlpatterns = patterns('',
    (r'latest/$', views.latest_books),
)
```

```
# latest_books.html (the template)
<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
</body></html>
```

Don't worry about the particulars of *how* this works just yet — we just want you to get a feel for the overall design. The main thing to note here is the *separation of concerns*:

- ✓ The models.py file contains a description of the database table, as a Python class. This is called a *model*. Using this class, you can create, retrieve, update, and delete records in your database using simple Python code rather than writing repetitive SQL statements.
- ✓ The views.py file contains the business logic for the page, in the latest_books() function. This function is called a *view*.
- ✓ The urls.py file specifies which view is called for a given URL pattern. In this case, the URL /latest/ will be handled by the latest_books() function.
- ✓ The latest_books.html is an HTML template that describes the design of the page.

Taken together, these pieces loosely follow the Model-View-Controller (MVC) design pattern. Simply put, MVC defines a way of developing software so that the code for defining and accessing data (the model) is separate from request routing logic (the controller), which in turn is separate from the user interface (the view).

A key advantage of such an approach is that components are *loosely coupled*. That is, each distinct piece of a Django-powered Web application has a single key purpose and can be changed independently without affecting the other pieces. For example, a developer can change the URL for a given part of the application without affecting the underlying implementation. A designer can change a page's HTML without having to touch the Python code that renders it. A database administrator can rename a database table and specify the change in a single place, rather than having to search and replace through a dozen files.

1.3 Django Evolution

Before we dive into more code, we should take a moment to explain Django's history. It's helpful to understand why the framework was created, because a knowledge of the history will put into context why Django works the way it does. If you've been building Web applications for a while, you're probably familiar with the problems in the CGI example we presented earlier. The classic Web developer's path goes something like this:

- ✓ Write a Web application from scratch.
- ✓ Write another Web application from scratch.
- ✓ Realize the application from step 1 shares much in common with the application from step 2.
- ✓ Refactor the code so that application 1 shares code with application 2.
- ✓ Repeat steps 2-4 several times.

✓ Realize you've invented a framework.

This is precisely how Django itself was created! Django grew organically from real-world applications written by a Web development team in Lawrence, Kansas. It was born in the fall of 2003, when the Web programmers at the Lawrence Journal-World newspaper, Adrian Holovaty and Simon Willison, began using Python to build applications. The World Online team, responsible for the production and maintenance of several local news sites, thrived in a development environment dictated by journalism deadlines. For the sites — including LJWorld.com, Lawrence.com, and KUsports.com — journalists (and management) demanded that features be added and entire applications be built on an intensely fast schedule, often with only days' or hours' notice. Thus, Adrian and Simon developed a time-saving Web development framework out of necessity — it was the only way they could build maintainable applications under the extreme deadlines.

In summer 2005, after having developed this framework to a point where it was efficiently powering most of World Online's sites, the World Online team, which now included Jacob Kaplan-Moss, decided to release the framework as open source software. They released it in July 2005 and named it Django, after the jazz guitarist Django Reinhardt.

Although Django is now an open source project with contributors across the planet, the original World Online developers still provide central guidance for the framework's growth, and World Online contributes other important aspects such as employee time, marketing materials, and hosting/bandwidth for the framework's Web site (http://www.djangoproject.com/).

This history is relevant because it helps explain two key matters. The first is Django's "sweet spot." Because Django was born in a news environment, it offers several features (particularly its admin interface, covered in Chapter 6) that are particularly well suited for "content" sites — sites like eBay, craigslist.org, and washingtonpost.com that offer dynamic, database-driven information. (Don't let that turn you off, though — although Django is particularly good for developing those sorts of sites, that doesn't preclude it from being an effective tool for building any sort of dynamic Web site. There's a difference between being *particularly effective* at something and being *ineffective* at other things.)

The second matter to note is how Django's origins have shaped the culture of its open source community. Because Django was extracted from real-world code, rather than being an academic exercise or commercial product, it is acutely focused on solving Web development problems that Django's developers themselves have faced — and continue to face. As a result, Django itself is actively improved on an almost daily basis. The framework's developers have a keen interest in making sure Django saves developers time, produces applications that are easy to maintain, and performs well under load. If nothing else, the developers are motivated by their own selfish desires to save themselves time and enjoy their jobs. (To put it bluntly, they eat their own dog food.)

1.4 Getting Help

One of the greatest benefits of Django is its kind and helpful user community. For help with any aspect of Django — from installation, to application design, to database design, to deployment — feel free to ask questions online.

- ✓ The django-users mailing list is where thousands of Django users hang out to ask and answer questions. Sign up for free at http://www.djangoproject.com/r/django-users.

The Django IRC channel is where Django users hang out to chat and help each other in real time. Join the fun by logging on to #django on the Freenode IRC network.