Introduction to Django

Table of Content

Why Django

As Websites grew and became more ambitious, it quickly became obvious that that situation was tedious, time-consuming, and ultimately untenable. A group of enterprising hackers at NCSA (the National Center for Supercomputing Applications, where Mosaic, the first graphical Web browser, was developed) solved this problem by letting the Web server spawn external programs that could dynamically generate HTML. They called this protocol the Common Gateway Interface, or CGI, and it changed the Web forever.

It's hard now to imagine what a revelation CGI must have been: instead of treating HTML pages as simple files on disk, CGI allows you to think of your pages as resources generated dynamically on demand. The development of CGI ushered in the first generation of dynamic Web sites.

However, CGI has its problems: CGI scripts need to contain a lot of repetitive "boilerplate" code, they make code reuse difficult, and they can be difficult for first-time developers to write and understand.

PHP fixed many of these problems, and it took the world by storm – it's now by far the most popular tool used to create dynamic Web sites, and dozens of similar languages and environments (ASP, JSP, etc.) followed PHP's design closely. PHP's major innovation is its ease of use: PHP code is simply embedded into plain HTML; the learning curve for someone who already knows HTML is extremely shallow.

But PHP has its own problems; its very ease of use encourages sloppy, repetitive, ill-conceived code. Worse, PHP does little to protect programmers from security vulnerabilities, and thus many PHP developers found themselves learning about security only once it was too late.

These and similar frustrations led directly to the development of the current crop of "third-generation" Web development frameworks. These frameworks – Django and Ruby on Rails appear to be the most popular these days – recognize that the Web's importance has escalated of late. With this new explosion of Web development comes yet another increase in ambition; Web developers are expected to do more and more every day.

Django was invented to meet these new ambitions. Django lets you build deep, dynamic, interesting sites in an extremely short time. Django is designed to let you focus on the fun, interesting parts of your job 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 on 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.

What Is a Web Framework?

Django is a prominent member of a new generation of Web frameworks – but what does that term mean, precisely? To answer that question, let's consider the design of a Web application written in Python without a framework. Throughout this reading, we'll take this approach of showing you basic ways of getting work done without shortcuts, in the hope that you'll recognize why shortcuts are so helpful. (It's also valuable to know how to get things done without shortcuts because shortcuts aren't always available. And most importantly, knowing why things work the way they do makes you a better Web developer.)

One of the simplest, most direct ways to build a Python Web app from scratch is to use the Common Gateway Interface (CGI) standard, which was a popular technique circa 1998. Here's a high-level explanation of how it works: just create a Python script that outputs HTML, then save the script to a Web server with a ".cgi" extension and visit the page in your Web browser. That's it.

Here's an example Python CGI script that displays the ten most recently published books from a database. Don't worry about syntax details; just get a feel for the basic things it's doing:

```
#!/usr/bin/env python

import MySQLdb

print "Content-Type: text/html\n"
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()
```

First, to fulfill the requirements of CGI, this code prints a "Content-Type" line, followed by a blank line. It prints some introductory HTML, connects to a database and runs a query to retrieve the names of the latest ten books. Looping over those books, it generates an HTML list of the titles. Finally, it prints the closing HTML and closes the database connection.

With a one-off page like 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 everything 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 that ends with ".cgi", upload that file to a Web server, and visit that page with a browser.

But despite its simplicity, this approach has a number of problems and annoyances. Ask yourself these questions:

- What happens when multiple parts of your application need to connect to the database? Surely that database-connecting code shouldn't need to be duplicated in each individual CGI script. 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? One wrong character could crash the entire application. Ideally, the logic of the page – the retrieval of book titles 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.

Let's dive in with a quick example that demonstrates the difference between the previous approach and a Web framework's approach. Here's how you might write the previous CGI code using Django. The first thing to note is that that we split it over four Python files (models.py, views.py, urls.py) and an HTML template (latest_books.html):

```
# models.py (the database tables)

from django.db import models

class Book(models.Model):
    name = models.CharField(max_length=50)
    pub_date = models.DateField()
```

```
# views.py (the business logic)

from django.shortcuts import render
from models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render(request, '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>
```

Again, don't worry about the particulars of syntax; just 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, represented by a Python class. This class is called a model. Using it, 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. The latest_books() 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. In other words, if your domain is example.com, any visit to the URL http://example.com/latest/ will call the latest_books() function.
- The latest_books.html file is an HTML template that describes the design of the page. It uses a template language with basic logic statements – e.g., {% for book in book_list %}.

Taken together, these pieces loosely follow a pattern called Model-View-Controller (MVC). Simply put, MVC is 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. 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.

Views and URLconfs

Views

A Django view is the Python function that is called according to a particular URL mapping for an app page. Each Django view in your Web Framework app needs to be defined in the views.py file, which is a Python module that is located in your app's directory.

Essentially, a view is just a Python function that takes an HttpRequest as its first parameter and returns an instance of HttpResponse. In order for a Python function to be a Django view, it must do these two things.

URLconfs

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations. There's no .php or .cgi required, and certainly none of that 0,2097,1-1-1928,00 nonsense.

A URLconf is like a table of contents for your Django-powered Web site. Basically, it's a mapping between URLs and the view functions that should be called for those URLs. It's how you tell Django, "For this URL, call this code, and for that URL, call that code." For example, "When somebody visits the URL /foo/, call the view function foo_view(), which lives in the Python module views.py."

How Django processes a request

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

1. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL.

2. Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function. The view gets passed the following arguments:
3. If no regex matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view.

Example

Here's a sample view and the corresponding URLconf:

```
#views.py
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Let's step through the changes we've made to views.py to accommodate the current_datetime view.

- We've added an import datetime to the top of the module, so we can calculate dates.
- The new current_datetime function calculates the current date and time, as a datetime.datetime object, and stores that as the local variable now.
- The second line of code within the view constructs an HTML response using Python's "format-string" capability. The %s within the string is a placeholder, and the percent sign after the string means "Replace the %s in the preceding string with the value of the variable now." The now variable is technically a datetime.datetime object, not a string, but the %s format character converts it to its string representation, which is something like "2008-12-13 14:09:39.002731". This will result in an HTML string such as "<html><body>It is now 2008-12-13 14:09:39.002731.</body></html>".
- (Yes, our HTML is invalid, but we're trying to keep the example simple and short.)
- Finally, the view returns an HttpResponse object that contains the generated response.

After adding that to views.py, add the URLpattern to urls.py to tell Django which URL should handle this view. Something like /time/ would make sense:

```
# urls.py (the URL configuration)

from django.conf.urls.defaults import patterns, include, url
from mysite.views import hello, current_datetime

urlpatterns = patterns('',
    url(r'^time/$', current_datetime),
)
```

Now, when we enter the page /time/, the URLconfs will point us to the `current_datetime` function in views, and display an HTML page.

URLconfs and Loose Coupling

Now's a good time to highlight a key philosophy behind URLconfs and behind Django in general: the principle of loose coupling. Simply put, loose coupling is a software development approach that values the importance of making pieces interchangeable. If two pieces of code are loosely coupled, then changes made to one of the pieces will have little or no effect on the other.

Django's URLconfs are a good example of this principle in practice. In a Django web application, the URL definitions and the view functions they call are loosely coupled; that is, the decision of what the URL should be for a given function, and the implementation of the function itself, reside in two separate places. This lets you switch out one piece without affecting the other.

For example, consider our **`current_datetime`** view. If we wanted to change the URL for the application – say, to move it from `/time/` to `/currenttime/` – we could make a quick change to the URLconf, without having to worry about the view itself. Similarly, if we wanted to change the view function – altering its logic somehow – we could do that without affecting the URL to which the function is bound.

Furthermore, if we wanted to expose the **`current_datetime`** functionality at several URLs, we could easily take care of that by editing the URLconf, without having to touch the view code. In this example, our **`current_datetime`** is available at two URLs. It's a contrived example, but this technique can come in handy:

```
# urls.py (the URL configuration)
```

```
urlpatterns = patterns('',
url(r'^hello/$', hello),
url(r'^time/$', current_datetime), url(r'^anothertimepage/$',
current_datetime),

)
```

URLconfs and views are loose coupling in action.

Templates

Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted. Django's template engine provides a powerful mini-language for defining the user-facing layer of your application, encouraging a clean separation of application and presentation logic. Templates can be maintained by anyone with an understanding of HTML; no knowledge of Python is required.

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, etc.). A template contains variables, which get replaced with values when the template is evaluated, and tags, which control the logic of the template. Below is a minimal template that illustrates a few basics.

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

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>

{% endfor %}

{% endblock %}
```

Understanding Templating Language

Variables look like this: **`{{ variable }}`**. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore ("_"). The dot (".") also appears in variable sections, although that has a special meaning, as indicated below. Importantly, you cannot have spaces or punctuation characters in variable names.

In the above example, **`{{ section.title }}`** will be replaced with the title attribute of the section object. If you use a variable that doesn't exist, the template system will insert the value of the `string_if_invalid` option, which is set to '' (the empty string) by default.

Tags look like this: **`{% tag %}`**. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables. Some tags require beginning and ending tags (i.e. **`{% tag %}`** ... tag contents ... **`{% endtag %}`**). In the example above, you can see that we used a **`for`** tag.

Template Inheritance

The most powerful – and thus the most complex – part of Django's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines blocks that child templates can override. The block and extends blocks set up template inheritance, a powerful way of cutting down on "boilerplate" in templates.

You can read more about templates here.

Models

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses django.db.models.Model.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API for several different DBMS systems, including SQLites, MySQL, and Postgres.

Example

This example model defines a Person, which has a `first_name` and `last_name`:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` and `last_name` are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above Person model would create a database table like this:

```
CREATE TABLE myapp_person (
    `id` auto_increment NOT NULL PRIMARY KEY,
    `first_name` varchar(30) NOT NULL,
    `last_name` varchar(30) NOT NULL
);
```

Some technical notes:

- The name of the table, **`myapp_person`**, is automatically derived from some model metadata but can be overridden.
- An id field is added automatically, but this behavior can be overridden.
- The `CREATE TABLE SQL` in this example is formatted using MySQL syntax, but it's worth noting Django uses SQL tailored to the database backend specified in your settings file.

Additional Links

The Django Book (http://www.djangobook.com/en/2.0/index.html)

Django Project (https://www.djangoproject.com/)