Django

Documentation

Django at a glance

Because Django was developed in a fast-paced newsroom environment, it was designed to make common Web-development tasks fast and easy. Here's an informal overview of how to write a database-driven Web app with Django.

The goal of this document is to give you enough technical specifics to understand how Django works, but this isn't intended to be a tutorial or reference – but we've got both! When you're ready to start a project, you can start with the tutorial or dive right into more detailed documentation.

Design your model

Although you can use Django without a database, it comes with an object-relational mapper in which you describe your database layout in Python code.

The data-model syntax offers many rich ways of representing your models – so far, it's been solving many years' worth of database-schema problems. Here's a quick example:

mysite/news/models.py

```
from django.db import models

class Reporter(models.Model):
    full_name = models.CharField(max_length=70)

    def __str__(self):              # __unicode__ on Python 2
        return self.full_name

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)

    def __str__(self):              # __unicode__ on Python 2
        return self.headline
```

Install it

Next, run the Django command-line utility to create the database tables automatically:

```
$ python manage.py migrate
```

The **`migrate`** command looks at all your available models and creates tables in your database for whichever tables don't already exist, as well as optionally providing much richer schema control.

Enjoy the free API

With that, you've got a free, and rich, Python API to access your data. The API is created on the fly, no code generation necessary:

```
# Import the models we created from our "news" app
>>> from news.models import Reporter, Article

# No reporters are in the system yet.
>>> Reporter.objects.all()
<QuerySet []>

# Create a new Reporter.
>>> r = Reporter(full_name='John Smith')

# Save the object into the database. You have to call save() explicitly.
>>> r.save()

# Now it has an ID.
>>> r.id
1

# Now the new reporter is in the database.
>>> Reporter.objects.all()
<QuerySet [<Reporter: John Smith>]>

# Fields are represented as attributes on the Python object.
>>> r.full_name
'John Smith'

# Django provides a rich database lookup API.
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist.

# Create an article.
>>> from datetime import date
>>> a = Article(pub_date=date.today(), headline='Django is cool',
...     content='Yeah.', reporter=r)
>>> a.save()

# Now the article is in the database.
>>> Article.objects.all()
<QuerySet [<Article: Django is cool>]>

# Article objects get API access to related Reporter objects.
>>> r = a.reporter
>>> r.full_name
'John Smith'

# And vice versa: Reporter objects get API access to Article objects.
>>> r.article_set.all()
<QuerySet [<Article: Django is cool>]>

# The API follows relationships as far as you need, performing efficient
# JOINs for you behind the scenes.
# This finds all articles by a reporter whose name starts with "John".
>>> Article.objects.filter(reporter__full_name__startswith='John')
<QuerySet [<Article: Django is cool>]>

# Change an object by altering its attributes and calling save().
>>> r.full_name = 'Billy Goat'
>>> r.save()

# Delete an object with delete().
>>> r.delete()
```

Once your models are defined, Django can automatically create a professional, production ready administrative interface – a website that lets authenticated users add, change and delete objects. It's as easy as registering your model in the admin site:

mysite/news/models.py

```
from django.db import models

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)
```

mysite/news/admin.py

```
from django.contrib import admin

from . import models

admin.site.register(models.Article)
```

The philosophy here is that your site is edited by a staff, or a client, or maybe just you – and you don't want to have to deal with creating backend interfaces just to manage content.

One typical workflow in creating Django apps is to create models and get the admin sites up and running as fast as possible, so your staff (or clients) can start populating data. Then, develop the way data is presented to the public.

Design your URLs

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django encourages beautiful URL design and doesn't put any cruft in URLs, like **`.php`** or **`.asp`**.

To design URLs for an app, you create a Python module called a URLconf. A table of contents for your app, it contains a simple mapping between URL patterns and Python callback functions. URLconfs also serve to decouple URLs from Python code.

Here's what a URLconf might look like for the **`Reporter`**/**`Article`** example above:

mysite/news/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

The code above maps URLs, as simple regular expressions, to the location of Python callback functions ("views"). The regular expressions use parenthesis to "capture" values from the URLs. When a user requests a page, Django runs through each pattern, in order, and stops at the first one that matches the requested URL. (If none of them matches, Django calls a special-case 404 view.) This is blazingly fast, because the regular expressions are compiled at load time.

Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function. Each view gets passed a request object – which contains request metadata – and the values captured in the regex.

For example, if a user requested the URL "/articles/2005/05/39323/", Django would call the function **`news.views.article_detail(request, '2005', '05', '39323')`**.

Write your views

Each view is responsible for doing one of two things: Returning an **`HttpResponse`** object containing the content for the requested page, or raising an exception such as **`Http404`**. The rest is up to you.

Generally, a view retrieves data according to the parameters, loads a template and renders the template with the retrieved data. Here's an example view for **`year_archive`** from above:

mysite/news/views.py

```
from django.shortcuts import render

from .models import Article

def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    context = {'year': year, 'article_list': a_list}
    return render(request, 'news/year_archive.html', context)
```

This example uses Django's template system, which has several powerful features but strives to stay simple enough for non-programmers to use.

Design your templates

The code above loads the **news/year_archive.html** template.

Django has a template search path, which allows you to minimize redundancy among templates. In your Django settings, you specify a list of directories to check for templates with **DIRS**. If a template doesn't exist in the first directory, it checks the second, and so on.

Let's say the **news/year_archive.html** template was found. Here's what that might look like:

mysite/news/templates/news/year_archive.html

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

{% block title %}Articles for {{ year }}{% endblock %}

{% block content %}
<h1>Articles for {{ year }}</h1>

{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>By {{ article.reporter.full_name }}</p>
    <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

Variables are surrounded by double-curly braces. **{{ article.headline }}** means "Output the value of the article's headline attribute." But dots aren't used only for attribute lookup. They also can do dictionary-key lookup, index lookup and function calls.

Note **{{ article.pub_date|date:"F j, Y" }}** uses a Unix-style "pipe" (the "|" character). This is called a template filter, and it's a way to filter the value of a variable. In this case, the date filter formats a Python datetime object in the given format (as found in PHP's date function).

You can chain together as many filters as you'd like. You can write custom template filters. You can write custom template tags, which run custom Python code behind the scenes.

Finally, Django uses the concept of "template inheritance". That's what the **{% extends "base.html" %}** does. It means "First load the template called 'base', which has defined a bunch of blocks, and fill the blocks with the following blocks." In short, that lets you dramatically cut down on redundancy in templates: each template has to define only what's unique to that template.

Here's what the "base.html" template, including the use of static files, might look like:

mysite/templates/base.html

```
{% load static %}
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <img src="{% static "images/sitelogo.png" %}" alt="Logo" />
    {% block content %}{% endblock %}
</body>
</html>
```

Simplistically, it defines the look-and-feel of the site (with the site's logo), and provides "holes" for child templates to fill. This makes a site redesign as easy as changing a single file – the base template.

It also lets you create multiple versions of a site, with different base templates, while reusing child templates. Django's creators have used this technique to create strikingly different mobile versions of sites – simply by creating a new base template.

Note that you don't have to use Django's template system if you prefer another system. While Django's template system is particularly well-integrated with Django's model layer, nothing forces you to use it. For that matter, you don't have to use Django's database API, either. You can use another database abstraction layer, you can read XML files, you can read files off disk, or anything you want. Each piece of Django – models, views, templates – is decoupled from the next.

This is just the surface

This has been only a quick overview of Django's functionality. Some more useful features:

- A caching framework that integrates with memcached or other backends.
- A syndication framework that makes creating RSS and Atom feeds as easy as writing a small Python class.
- More sexy automatically-generated admin features – this overview barely scratched the surface.

The next obvious steps are for you to download Django, read the tutorial and join the community. Thanks for your interest!

Getting started

Quick install guide