Django

Documentation

Writing your first Django app, part 1

Let's learn by example.

Throughout this tutorial, we'll walk you through the creation of a basic poll application.

It'll consist of two parts:

- A public site that lets people view polls and vote in them.
- An admin site that lets you add, change, and delete polls.

We'll assume you have Django installed already. You can tell Django is installed and which version by running the following command:

```
$ python -c "import django; print(django.get_version())"
```

If Django is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named django".

This tutorial is written for Django 1.9 and Python 3.4 or later. If the Django version doesn't match, you can refer to the tutorial for your version of Django by using the version switcher at the bottom right corner of this page, or update Django to the newest version. If you are still using Python 2.7, you will need to adjust the code samples slightly, as described in comments.

See How to install Django for advice on how to remove older versions of Django and install a newer one.

Where to get help:

If you're having trouble going through this tutorial, please post a message to django-users or drop by #django on irc.freenode.net to chat with other Django users who might be able to help.

Creating a project

If this is your first time using Django, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Django project – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, **cd** into a directory where you'd like to store your code, then run the following command:

```
$ django-admin startproject mysite
```

This will create a **mysite** directory in your current directory. If it didn't work, see Problems running django-admin.

Note

You'll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like **django** (which will conflict with Django itself) or **test** (which conflicts with a built-in Python package).

Where should this code live?

If your background is in plain old PHP (with no use of modern frameworks), you're probably used to putting code under the Web server's document root (in a place such as **/var/www**). With Django, you don't do that. It's not a good idea to put any of this Python code within your Web server's document root, because it risks the possibility that people may be able to view your code over the Web. That's not good for security.

Put your code in some directory **outside** of the document root, such as **/home/mycode**.

Let's look at what **startproject** created:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

These files are:

- The outer **mysite/** root directory is just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- **manage.py**: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about **manage.py** in django-admin and manage.py.
- The inner **mysite/** directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. **mysite.urls**).
- **mysite/__init__.py**: An empty file that tells Python that this directory should be considered a Python package. If you're a Python beginner, read more about packages in the official Python docs.
- **mysite/settings.py**: Settings/configuration for this Django project. Django settings will tell you all about how settings work.
- **mysite/urls.py**: The URL declarations for this Django project; a "table of contents" of your Django-powered site. You can read more about URLs in URL dispatcher.
- **mysite/wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project. See How to deploy with WSGI for more details.

The development server

Let's verify your Django project works. Change into the outer **mysite** directory, if you haven't already, and run the following commands:

```
$ python manage.py runserver
```

You'll see the following output on the command line:

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.

May 13, 2016 - 15:50:53
Django version 1.9, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Note

Ignore the warning about unapplied database migrations for now; we'll deal with the database shortly.

You've started the Django development server, a lightweight Web server written purely in Python. We've included this with Django so you can develop things rapidly, without having to deal with configuring a production server – such as Apache – until you're ready for production.

Now's a good time to note: **don't** use this server in anything resembling a production environment. It's intended only for use while developing. (We're in the business of making Web frameworks, not Web servers.)

Now that the server's running, visit http://127.0.0.1:8000/ with your Web browser. You'll see a "Welcome to Django" page, in pleasant, light-blue pastel. It worked!

Changing the port

By default, the **`runserver`** command starts the development server on the internal IP at port 8000.

If you want to change the server's port, pass it as a command-line argument. For instance, this command starts the server on port 8080:

```
$ python manage.py runserver 8080
```

If you want to change the server's IP, pass it along with the port. So to listen on all public IPs (useful if you want to show off your work on other computers on your network), use:

```
$ python manage.py runserver 0.0.0.0:8000
```

Full docs for the development server can be found in the **`runserver`** reference.

Automatic reloading of `runserver`

The development server automatically reloads Python code for each request as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

Creating the Polls app

Now that your environment – a "project" – is set up, you're set to start doing work.

Each application you write in Django consists of a Python package that follows a certain convention. Django comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories.

Projects vs. apps

What's the difference between a project and an app? An app is a Web application that does something – e.g., a Weblog system, a database of public records or a simple poll app. A project is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.

Your apps can live anywhere on your Python path. In this tutorial, we'll create our poll app right next to your **`manage.py`** file so that it can be imported as its own top-level module, rather than a submodule of **`mysite`**.

To create your app, make sure you're in the same directory as **`manage.py`** and type this command:

```
$ python manage.py startapp polls
```

That'll create a directory **`polls`**, which is laid out like this:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

This directory structure will house the poll application.

Write your first view

Let's write the first view. Open the file **polls/views.py** and put the following Python code in it:

polls/views.py

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

This is the simplest view possible in Django. To call the view, we need to map it to a URL - and for this we need a URLconf.

To create a URLconf in the polls directory, create a file called **urls.py**. Your app directory should now look like:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

In the **polls/urls.py** file include the following code:

polls/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

The next step is to point the root URLconf at the **polls.urls** module. In **mysite/urls.py**, add an import for **django.conf.urls.include** and insert an **include()** in the **urlpatterns** list, so you have:

mysite/urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', admin.site.urls),
]
```

The **include()** function allows referencing other URLconfs. Note that the regular expressions for the **include()** function doesn't have a **$** (end-of-string match character) but rather a trailing slash. Whenever Django encounters **include()**, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

The idea behind **include()** is to make it easy to plug-and-play URLs. Since polls are in their own URLconf (**polls/urls.py**), they can be placed under "/polls/", or under "/fun_polls/", or under "/content/polls/", or any other path root, and the app will still work.

When to use include()

You should always use **include()** when you include other URL patterns. **admin.site.urls** is the only exception to this.

Doesn't match what you see?

If you're seeing **`include(admin.site.urls)`** instead of just **`admin.site.urls`**, you're probably using a version of Django that doesn't match this tutorial version. You'll want to either switch to the older tutorial or the newer Django version.

You have now wired an **`index`** view into the URLconf. Lets verify it's working, run the following command:

```
$ python manage.py runserver
```

Go to http://localhost:8000/polls/ in your browser, and you should see the text "*Hello, world. You're at the polls index.*", which you defined in the **`index`** view.

The **`url()`** function is passed four arguments, two required: **`regex`** and **`view`**, and two optional: **`kwargs`**, and **`name`**. At this point, it's worth reviewing what these arguments are for.

`url()` argument: regex

The term "regex" is a commonly used short form meaning "regular expression", which is a syntax for matching patterns in strings, or in this case, url patterns. Django starts at the first regular expression and makes its way down the list, comparing the requested URL against each regular expression until it finds one that matches.

Note that these regular expressions do not search GET and POST parameters, or the domain name. For example, in a request to **`https://www.example.com/myapp/`**, the URLconf will look for **`myapp/`**. In a request to **`https://www.example.com/myapp/?page=3`**, the URLconf will also look for **`myapp/`**.

If you need help with regular expressions, see Wikipedia's entry and the documentation of the **`re`** module. Also, the O'Reilly book "Mastering Regular Expressions" by Jeffrey Friedl is fantastic. In practice, however, you don't need to be an expert on regular expressions, as you really only need to know how to capture simple patterns. In fact, complex regexes can have poor lookup performance, so you probably shouldn't rely on the full power of regexes.

Finally, a performance note: these regular expressions are compiled the first time the URLconf module is loaded. They're super fast (as long as the lookups aren't too complex as noted above).

`url()` argument: view

When Django finds a regular expression match, Django calls the specified view function, with an **`HttpRequest`** object as the first argument and any "captured" values from the regular expression as other arguments. If the regex uses simple captures, values are passed as positional arguments; if it uses named captures, values are passed as keyword arguments. We'll give an example of this in a bit.

`url()` argument: kwargs

Arbitrary keyword arguments can be passed in a dictionary to the target view. We aren't going to use this feature of Django in the tutorial.

`url()` argument: name

Naming your URL lets you refer to it unambiguously from elsewhere in Django especially templates. This powerful feature allows you to make global changes to the url patterns of your project while only touching a single file.

When you're comfortable with the basic request and response flow, read part 2 of this tutorial to start working with the database.

‹ Quick install guide

Writing your first Django app, part 2 ›

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

Language: **en**

Documentation version: **1.9**