O'REILLY®

Lightweight Django

USING REST, WEBSOCKETS & BACKBONE

Julia Elman & Mark Lavin

Lightweight Django

by Julia Elman and Mark Lavin

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meghan Blanchette
Production Editor: Colleen Lobner
Copyeditor: Rachel Monaghan
Proofreader: Sonia Saruba

Indexer: Wendy Catalano
Cover Designer: Ellie Volckhausen
Interior Designer: David Futato
Illustrator: Rebecca Demarest

November 2014: First Edition

Revision History for the First Edition:

2014-10-24: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491945940* for release details.

ISBN: 978-1-491-94594-0

LSI

Table of Contents

CHAPTER 1
The World's Smallest Django Project

How many of our journeys into using Django have begun with the official polls tutorial? For many it seems like a rite of passage, but as an introduction to Django it is a fairly daunting task. With various commands to run and files to generate, it is even harder to tell the difference between a project and an application. For new users wanting to start building applications with Django, it begins to feel far too "heavy" as an option for a web framework. What are some ways we can ease these new users' fears to create a clean and simple start?

Let's take a moment to consider the recommended tasks for starting a Django project. The creation of a new project generally starts with the `startproject` command. There is no real magic to what this command does; it simply creates a few files and directories.

While the `startproject` command is a useful tool, it is not required in order to start a Django project. You are free to lay out your project however you like based on what you want to do. For larger projects, developers benefit from the code organization provided by the `startproject` command. However, the convenience of this command shouldn't stop you from understanding what it does and why it is helpful.

In this chapter we'll lay out an example of how to create a simple project using Django's basic building blocks. This lightweight "Hello World" project will create a simple Django application using a single-file approach.

Hello Django

Building a "Hello World" example in a new language or framework is a common first project. We've seen this simple starter project example come out of the Flask community to display how lightweight it is as a microframework.

In this chapter, we'll start by using a single *hello.py* file. This file will contain all of the code needed to run our Django project. In order to have a full working project, we'll

need to create a view to serve the root URL and the necessary settings to configure the Django environment.

Creating the View

Django is referred to as a *model-template-view* (MTV) framework. The view portion typically inspects the incoming HTTP request and queries, or constructs, the necessary data to send to the presentation layer.

In our example *hello.py* file, let's create a simple way to execute a "Hello World" response.

```
from django.http import HttpResponse


def index(request):
    return HttpResponse('Hello World')
```

In a larger project, this would typically be in a *views.py* file inside one of your apps. However, there is no requirement for views to live inside of apps. There is also no requirement that views live in a file called *views.py*. This is purely a matter of convention, but not a requirement on which to base our project's structure.

The URL Patterns

In order to tie our view into the site's structure, we'll need to associate it with a URL pattern. For this example, the server root can serve the view on its own. Django associates views with their URL by pairing a regular expression to match the URL and any callable arguments to the view. The following is an example from *hello.py* of how we make this connection.

```
from django.conf.urls import url
from django.http import HttpResponse


def index(request):
    return HttpResponse('Hello World')


urlpatterns = (
    url(r'^$', index),
)
```

Now this file combines both a typical *views.py* file and the root *urls.py* file. Again, it is worth noting that there is no requirement for the URL patterns to be included in a *urls.py* file. They can live in any importable Python module.

Let's move on to our Django settings and the simple lines we'll need to make our project runnable.

The Settings

Django settings detail everything from database and cache connections to internationalization features and static and uploaded resources. For many developers just getting started, the settings in Django are a major point of confusion. While recent releases have worked to trim down the default settings' file length, it can still be overwhelming.

This example will run Django in debugging mode. Beyond that, Django merely needs to be configured to know where the root URLs can be found and will use the value defined by the `urlpatterns` variable in that module. In this example from *hello.py*, the root URLs are in the current module and will use the `urlpatterns` defined in the previous section.

```
from django.conf import settings


settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)
...
```

This example includes a nonrandom `SECRET_KEY` setting, which should not be used in a production environment. A secret key must be generated for the default session and cross-site request forgery (CSRF) protection. It is important for any production site to have a random `SECRET_KEY` that is kept private. To learn more, go to the documentation at *https://docs.djangoproject.com/en/1.7/topics/signing/*.

We need to configure the settings before making any additional imports from Django, as some parts of the framework expect the settings to be configured before they are imported. Normally, this wouldn't be an issue since these settings would be included in their own *settings.py* file. The file generated by the default `startproject` command would also include settings for things that aren't used by this example, such as the internationalization and static resources.

Running the Example

Let's take a look at what our example looks like during `runserver`. A typical Django project contains a *manage.py* file, which is used to run various commands such as creating database tables and running the development server. This file itself is a total of 10 lines of code. We'll be adding in the relevant portions of this file into our *hello.py* to create the same abilities *manage.py* has:

```
import sys


from django.conf import settings


settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)


from django.conf.urls import url
from django.http import HttpResponse


def index(request):
    return HttpResponse('Hello World')


urlpatterns = (
    url(r'^$', index),
)


if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

Now you can start the example in the command line:

```
hostname $ python hello.py runserver
Performing system checks...

System check identified no issues (0 silenced).
August 06, 2014 - 19:15:36
Django version 1.7c2, using settings None
```

```
Starting development server at http://7.0.0.1:8000/
Quit the server with CONTROL-C.
```

and visit *http://localhost:8000/* in your favorite browser to see "Hello World," as seen in Figure 1-1.

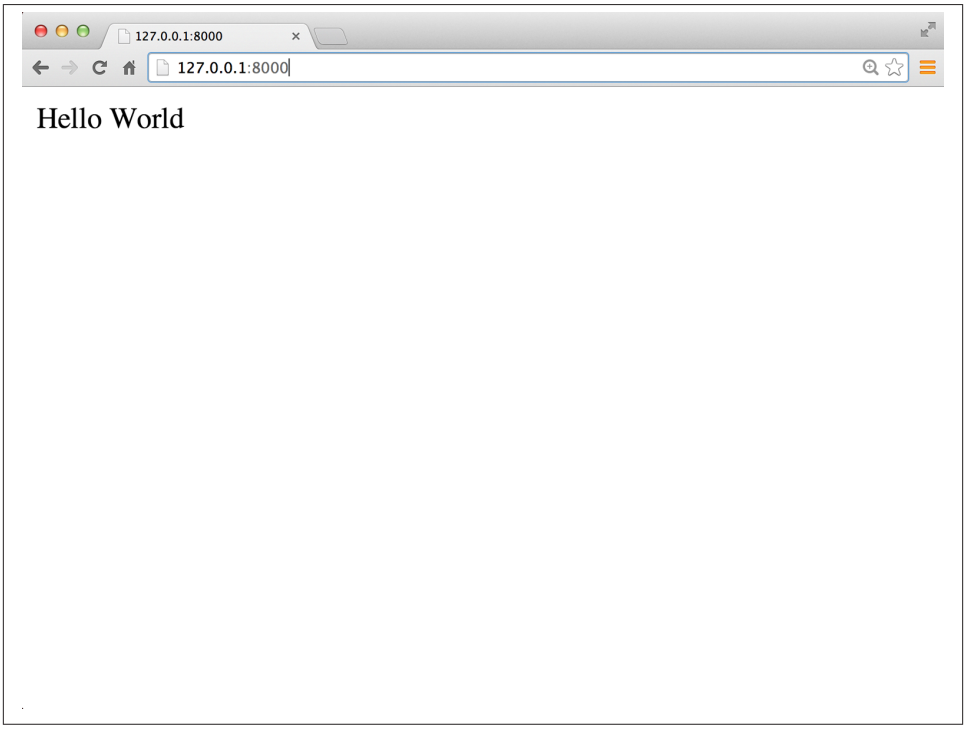

Figure 1-1. Hello World

Now that we have a very basic file structure in place, let's move on to adding more elements to serve up our files.

Improvements

This example shows some of the fundamental pieces of the Django framework: writing views, creating settings, and running management commands. At its core, Django is a Python framework for taking incoming HTTP requests and returning HTTP responses. What happens in between is up to you.

Django also provides additional utilities for common tasks involved in handling HTTP requests, such as rendering HTML, parsing form data, and persisting session state. While not required, it is important to understand how these features can be used in

your application in a lightweight manner. By doing so, you gain a better understanding of the overall Django framework and true capabilities.

WSGI Application

Currently, our "Hello World" project runs through the `runserver` command. This is a simple server based on the socket server in the standard library. It has helpful utilities for local development such as auto–code reloading. While it is convenient for local development, `runserver` is not appropriate for production deployment security.

The Web Server Gateway Interface (WSGI) is the specification for how web servers communicate with application frameworks such as Django, and was defined by PEP 333 and improved in PEP 3333. There are numerous choices for web servers that speak WSGI, including Apache via mod_wsgi, Gunicorn, uWSGI, CherryPy, Tornado, and Chaussette.

Each of these servers needs a properly defined WSGI application to be used. Django has an easy interface for creating this application through `get_wsgi_application`.

```
...
from django.conf.urls import url
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse
...
application = get_wsgi_application()

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

This would normally be contained within the *wsgi.py* file created by the `startproject` command. The name *application* is merely a convention used by most WSGI servers; each provides configuration options to use a different name if needed.

Now our simple Django project is ready for the WSGI server. Gunicorn is a popular choice for a pure-Python WSGI application server; it has a solid performance record, is easy to install, and also runs on Python 3. Gunicorn can be installed via the Python Package Index (`pip`).

```
hostname $ pip install gunicorn
```

Once Gunicorn is installed, you can run it fairly simply by using the `gunicorn` command.

```
hostname $ gunicorn hello --log-file=-
[2014-08-06 19:17:26 -0400] [37043] [INFO] Starting gunicorn 19.1.1
[2014-08-06 19:17:26 -0400] [37043] [INFO]
    Listening at: http://127.0.0.1:8000 (37043)
```

```
[2014-08-06 19:17:26 -0400] [37043] [INFO] Using worker: sync
[2014-08-06 19:17:26 -0400] [37046] [INFO] Booting worker with pid: 37046
```

As seen in the output, this example is running using Gunicorn version 19.1.1. The timestamps shown contain your time zone offset, which may differ depending on your locale. The process IDs for the arbiter and the worker will also be different.

As of R19, Gunicorn no longer logs to the console by default. Adding the `--log-file=-` option ensures that the output will be logged to the console. You can read more about Gunicorn settings at *http://docs.gunicorn.org/en/19.1/*.

As with `runserver` in Django, the server is listening on *http://127.0.0.1:8000/*. This works out nicely and makes an easier configuration for us to work with.

Additional Configuration

While Gunicorn is a production-ready web server, the application itself is not yet production ready, as `DEBUG` should never be enabled in production. As previously noted, the `SECRET_KEY` is also nonrandom and should be made random for additional security.

For more information on the security implications of the `DEBUG` and `SECRET_KEY` settings, please refer to the official Django documentation.

This leads to a common question in the Django community: how should the project manage different settings for development, staging, and production environments? Django's wiki contains a long list of approaches, and there are a number of reusable applications that aim to tackle this problem. A comparison of those applications can be found on Django Packages. While many of these options can be ideal in some cases, such as converting the *settings.py* into a package and creating modules for each environment, they do not line up well with our example's current single-file setup.

The Twelve Factor App is a methodology for building and deploying HTTP service applications. This methodology recommends separating configuration and code as well as storing configurations in environment variables. This makes the configuration easy to change on the deployment and makes the configuration OS-agnostic.

Let's apply this methodology to our *hello.py* example. There are only two settings that are likely to change between environments: `DEBUG` and `SECRET_KEY`.

```
import os
import sys

from django.conf import settings

DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY', os.urandom(32))

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)
```

As you may notice, the default for `DEBUG` is `True`, and the `SECRET_KEY` will be randomly generated each time the application is loaded if it is not set. That will work for this toy example, but if the application were using a piece of Django that requires the `SECRET_KEY` to remain stable, such as the signed cookies, this would cause the sessions to be frequently invalidated.

Let's examine how this translates to launching the application. To disable the `DEBUG` setting, we need to set the `DEBUG` environment variable to something other than `on`. In a UNIX-derivative system, such as Linux, OS X, or FreeBSD, environment variables are set on the command line with the `export` command. On Windows, you'd use `set`.

```
hostname $ export DEBUG=off
hostname $ python hello.py runserver
CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.
```

As you can see from the error, the `ALLOWED_HOSTS` setting isn't configured by our application. `ALLOWED_HOSTS` is used to validate incoming HTTP HOST header values and should be set to a list of acceptable values for the HOST. If the application is meant to serve `example.com`, then `ALLOWED_HOSTS` should allow only for clients that are requesting `example.com`. If the `ALLOWED_HOSTS` environment variable isn't set, then it will allow requests only for `localhost`. This snippet from *hello.py* illustrates.

```
import os
import sys

from django.conf import settings
```

```
DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY', os.urandom(32))

ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS', 'localhost').split(',')

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ALLOWED_HOSTS=ALLOWED_HOSTS,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)
```

With our `ALLOWED_HOSTS` variable set, we now have validation for our incoming HTTP HOST header values.

For a complete reference on the `ALLOWED_HOSTS` setting, see the official Django documentation.

Outside the development environment, the application might need to serve multiple hosts, such as `localhost` and `example.com`, so the configuration allows us to specify multiple hostnames separated by commas.

```
hostname $ export DEBUG=off
hostname $ export ALLOWED_HOSTS=localhost,example.com
hostname $ python hello.py runserver
...
[06/Aug/2014 19:45:53] "GET / HTTP/1.1" 200 11
```

This gives us a flexible means of configuration across environments. While it would be slightly more difficult to change more complex settings, such as `INSTALLED_APPS` or `MIDDLEWARE_CLASSES`, that is in line with the overall methodology, which encourages minimal differences between environments.

If you want to make complex changes between environments, you should take time to consider what impact that will have on the testability and deployment of the application.

We can reset DEBUG to the default by removing the environment variable from the shell or by starting a new shell.

```
hostname $ unset DEBUG
```

Reusable Template

So far this example has centered on rethinking the layout created by Django's `startproject` command. However, this command also allows for using a template to provide the layout. It isn't difficult to transform this file into a reusable template to start future projects using the same base layout.

A template for `startproject` is a directory or zip file that is rendered as a Django template when the command is run. By default, all of the Python source files will be rendered as a template. The rendering is passed `project_name`, `project_directory`, `secret_key`, and `docs_version` as the context. The names of the files will also be rendered with this context. To transform *hello.py* into a project template (*project_name/project_name.py*), the relevant parts of the file need to be replaced by these variables.

```
import os
import sys

from django.conf import settings

DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY', '{{ secret_key }}')

ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS', 'localhost').split(',')

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ALLOWED_HOSTS=ALLOWED_HOSTS,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)

from django.conf.urls import url
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse


def index(request):
    return HttpResponse('Hello World')
```

```
urlpatterns = (
    url(r'^$', index),
)


application = get_wsgi_application()


if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

Now let's save this file as *project_name.py* in a directory called *project_name*. Also, rather than using `os.urandom` for the `SECRET_KEY` default, this code will generate a random secret to be the default each time a new project is created. This makes the `SECRET_KEY` default stable at the project level while still being sufficiently random across projects.

To use the template with `startproject`, you can use the `--template` argument.

```
hostname $ django-admin.py startproject foo --template=project_name
```

This should create a *foo.py* inside a *foo* directory, which is now ready to run just like the original *hello.py*.

As outlined in this example, it is certainly possible to write and run a Django project without having to use the `startproject` command. The default settings and layout used by Django aren't appropriate for every project. The `--template` option for `startproject` can be used to either expand on these defaults or to trim them down, as you've seen in this chapter.

As with any Python project, there comes a point where organizing the code into multiple modules is an important part of the process. For a sufficiently focused site, with only a handful of URLs, our "Hello World" example may be a reasonable approach.

What is also interesting about this approach is that it isn't immediately obvious that Django has a templating engine or an object-relational mapper (ORM) built in. It is clear that you are free to choose whatever Python libraries you think best solve your problem. You no longer have to use the Django ORM, as the official tutorial might imply. Instead, you get to use the ORM if you want. The project in the next chapter will expand on this single-file example to provide a simple HTTP service and make use of more of the utilities that come with Django.