Creating a Django project

We've got our virtualenv set up inside our bugger directory and we've got Django installed inside our virtualenv. Before we actually create a proper Django project however, we'll first have a look at what makes a Django project.

In its purest form, a Django project is simply one or more modules, glued together using the project settings. In most — if not all — cases you'll write your own Python modules to add features to the project. Those modules are called "apps" in Django.

In Java with Maven

Let's compare this to a typical "pure" Java web application you can run in Tomcat or Jetty. Our project layout might contain the following:

```
pom.xml
src/main/java/com/acme/auth
src/main/java/com/acme/issues
src/main/webapp/WEB-INF/web.xml
src/test/java/com/acme/auth
src/test/java/com/acme/issues
```

Our Maven pom file defines our project structure with the dependencies of the application — things like Apache Commons libraries or Spring — and provides us with an easy way to run tests or create a WAR file from the sources.

In the `src/main/java` directory you'll add your Java sources, typically organized in packages like `com.acme.auth` and `com.acme.issues` according to the functionality the Java classes provide.

The `src/main/webapp` directory contains a `web.xml` file with the web app configuration and other web-related resources such as JSP, JavaScript, HTML and CSS files.

The `src/test/java` directory contains our tests to verify everything does what it's supposed to do.

If you'd like to reuse the `com.acme.auth` package in other projects, you'd move that package to a separate module or project resulting in a JAR file and then add that JAR as a dependency of your web app.

With Django

Assume you're porting this application to Django. Just like you organize related classes in packages in Java, you'll organize related code in Django in "apps"; in this case those two apps are called "auth" and "issues". The web app configuration you'd find in `web.xml` (or in a separate properties file, JNDI, doesn't matter really) is typically specified in at least two project-specific files: `settings.py` (database settings, caching, timezone,…) and `urls.py` (only the URL mapping, similar to servlet mappings).

Finally, most of the tools provided by Maven are either not needed, provided by virtualenv and pip or provided by Django's `manage.py` or `django-admin.py` scripts.

Don't worry if this all sounds a bit complicated. We're about to start a Django project and all pieces of the puzzle will slowly start falling into place.

startproject

We're ready to start our project. When starting a Maven-powered project, you'd either copy an existing `pom.xml` file or use a Maven archetype:

```
$ mvn archetype:generate -DgroupId=com.acme -DartifactId=bugger \
-DarchetypeArtifactId=maven-archetype-webapp
```

With Django, you use the startproject command like this:

```
(env) django-bugger $ django-admin.py startproject bugger
```

Note: we're inside the django-bugger directory in our `$PROJECTS` dir and are using our virtualenv `env`.

Let's have a look at what this command did.

```
(env) django-bugger $ tree -L 2
.
├── REQUIREMENTS
├── bugger
│   ├── bugger
│   └── manage.py
└── env
    ├── bin
    ├── include
    └── lib

6 directories, 2 files
```

`startproject` created a `bugger` directory to hold our apps and within that `bugger` directory is another directory called `bugger` which is our main app.

Yes, that's a lot of `bugger` directories and this *will* become confusing, so let's lay down some ground rules:

- `$PROJECTS/django-bugger` is our starting point. You'll now understand why we didn't call this directory `bugger`.
- `$PROJECTS/django-bugger/bugger` is our Django project directory.
- `$PROJECTS/django-bugger/bugger/bugger` is our main Django app.

Let's move to our bugger project directory and see what's going on.

```
(env) django-bugger $ cd bugger
(env) bugger $ tree -L 2
.
├── bugger
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py

1 directory, 5 files
```

As you can see above, our bugger app contains four Python files. The `__init__.py` file makes clear that the bugger app directory is actually a Python module. It's actually an empty file.

The `settings.py` file contains the Django configuration, `urls.py` contains our mapping of URLs to views and `wsgi.py` is an entry point for deploying this Django web application in a WSGI-compatible app server. More information on that will follow in later chapters.

In our bugger Django project we can find another file: `manage.py`. This is the entry point for commands issued related to the current Django project. It's actually nothing more than a way to invoke `django-admin.py` without having to tell it where your project settings live. For the record: your settings are specified in `settings.py`, which you can refer to as `bugger.settings`.

This means the following two commands are equivalent:

```
(env) bugger $ python manage.py startapp issues
# or
(env) bugger $ django-admin.py startapp issues --settings=bugger.settings
```

If you were using Maven, you could, with some extra configuration, start the web application through Maven in Jetty with this:

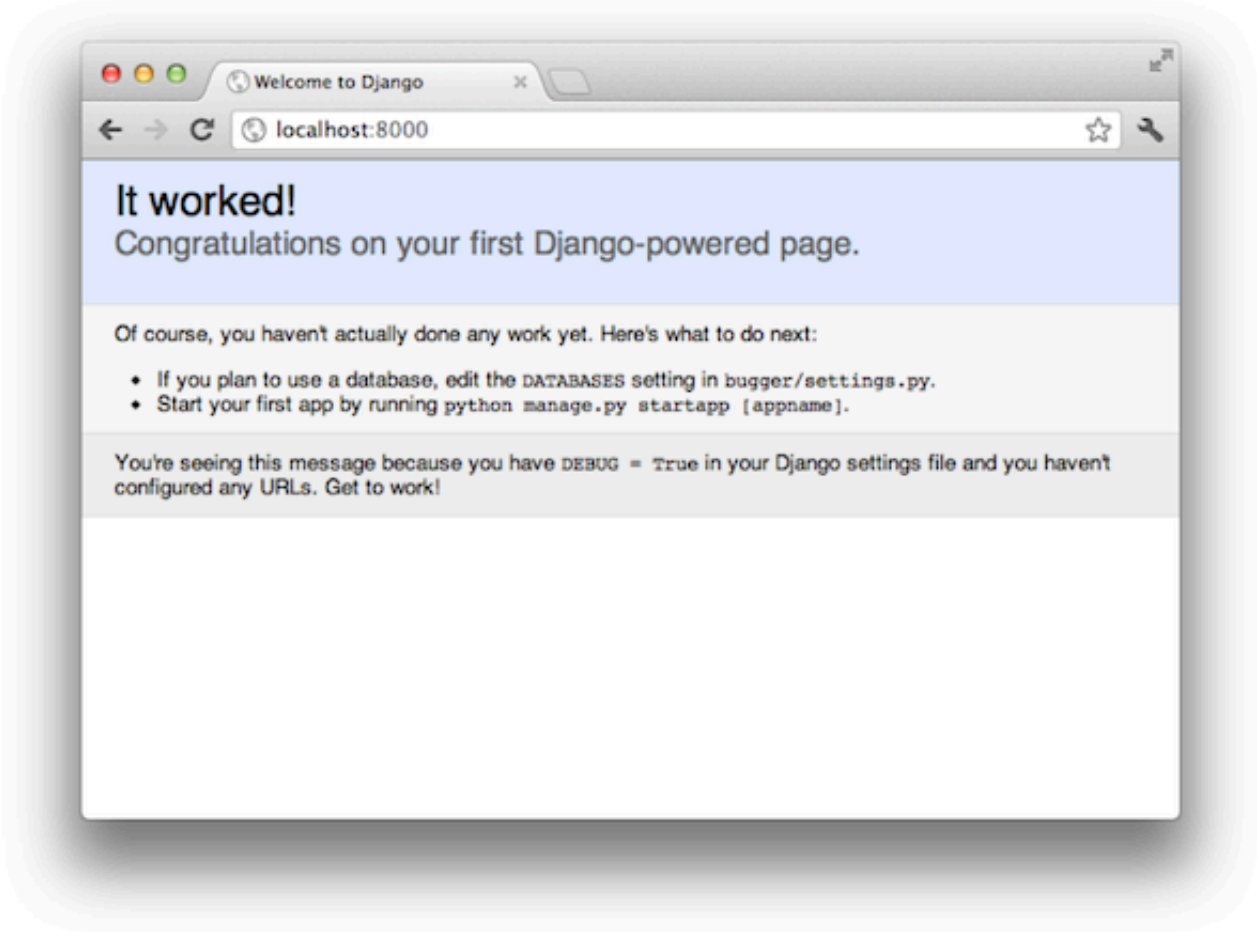
```
$ mvn jetty:run
```

Doing this in Django requires no extra configuration. Start a local dev server with the following:

```
(env) bugger $ python manage.py runserver
Validating models...

0 errors found
Django version 1.4, using settings 'bugger.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Open your browser and go to http://127.0.0.1:8000/ and you will be greeted by Django.



Ignore the instructions on this page for now and stop the server. First we're going to define what our Bugger application should do and examine how a Django application responds to HTTP requests.

If you've been reading all of this in one go, it might be time to go and have a cup of coffee, tea, scotch or any other beverage of choice. Give it some time to settle and look back at what you have learned and achieved. You already know how to define and install dependencies, how to easily create a new project and how to start the web server! And you haven't written a single line of code yet.