## Learn Python with Django


## 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 my site.

## Creating the catalog application

Next, run the following command to create the *catalog* application that will live inside our library project (this must be run in the same folder as your project's **manage.py**):

```
python manage.py startapp catalog
```

**Note**: the above command is for Linux/Mac OS X. On Windows the command should be: py -3 manage.py startapp catalog

The tool creates a new folder and populates it with files for the different parts of the application (shown in bold below). Most of the files are usefully named after their purpose (e.g. views should be stored in **views.py**, Models in **models.py**, tests in **tests.py**, administration site configuration in **admin.py**, application registration in **apps.py**) and contain some minimal boilerplate code for working with the associated objects.

The updated project directory should now look like this:

```
library/
    manage.py
    library/
    catalog/
        admin.py
        apps.py
        models.py
        tests.py
        views.py
        __init__.py
        migrations/
```

In addition we now have:

- A *migrations* folder, used to store "migrations" — files that allow you to automatically update your database as you modify your models.

- **__init__.py** — an empty file created here so that Django/Python will recognize the folder as a Python Package and allow you to use its objects within other parts of the project.

**Note**: Have you noticed what is missing from the files list above? While there is a place for your views and models, there is nowhere for you to put your URL mappings, templates, and static files. I'll show you how to create them further along (these aren't needed in every website but they are needed in this example).

## Registering the catalog application

Now that the application has been created we have to register it with the project so that it will be included when any tools are run (for example to add models to the database). Applications are registered by adding them to the INSTALLED_APPS list in the project settings.

Open the project settings file **library/library/settings.py** and find the definition for the INSTALLED_APPS list. Then add a new line at the end of the list, as shown in bold below.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'catalog.apps.CatalogConfig',
]
```

The new line specifies the application configuration object (CatalogConfig) that was generated for you in **/library/catalog/apps.py** when you created the application.

**Note**: You'll notice that there are already a lot of other INSTALLED_APPS (and MIDDLEWARE, further down in the settings file). These enable support for the Django administration site and as a result lots of the functionality it uses (including sessions, authentication, etc).

## Specifying the database

This is also the point where you would normally specify the database to be used for the project — it makes sense to use the same database for development and production where possible, in order to avoid minor differences in behaviour. You can find out about the different options in Databases (Django docs).

We'll use the SQLite database for this example, because we don't expect to require a lot of concurrent access on a demonstration database, and also because it requires no additional work to set up! You can see how this database is configured in **settings.py** (more information is also included below):

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Because we are using SQLite, we don't need to do any further setup here. Let's move on!

## Other project settings

The **settings.py** file is also used for configuring a number of other settings, but at this point you probably only want to change the TIME_ZONE — this should be made equal to a string from the standard List of tz database time zones (the TZ column in the table contains the values you want). Change your TIME_ZONE value to one of these strings appropriate for your time zone, for example:

```
TIME_ZONE = 'Europe/London'
```

There are two other settings you won't change now, but that you should be aware of:

- SECRET_KEY. This is a secret key that is used as part of Django's website security strategy. If you're not protecting this code in development, you'll need to use a different code (perhaps read from an environment variable or file) when putting it into production.
- DEBUG. This enables debugging logs to be displayed on error, rather than HTTP status code responses. This should be set to false on production as debug information is useful for attackers.

## Hooking up the URL mapper

The website is created with a URL mapper file (**urls.py**) in the project folder. While you can use this file to manage all your URL mappings, it is more usual to defer mappings to their associated application.

Open **library/library/urls.py** and note the instructional text which explains some of the ways to use the URL mapper.

```
"""library URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/2.0/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

The URL mappings are managed through the urlpatterns variable, which is a Python *list* of path() functions. Each path() function either associates a URL pattern to a *specific view*, which will be displayed when the pattern is matched, or with another list of URL pattern testing code (in this second case, the pattern becomes the "base URL" for patterns defined in the target module). The urlpatterns list initially defines a single function that maps all URLs with the pattern *admin/* to the module admin.site.urls , which contains the Administration application's own URL mapping definitions.

**Note**: The route in path() is a string defining a URL pattern to match. This string might include a named variable (in angle brackets), e.g. 'catalog/<id>/'. This pattern will match a URL like ***/catalog/****any_chars****/*** and pass *any_chars* to the view as a string with parameter name id). We discuss path methods and route patterns further in later topics.

Add the lines below to the bottom of the file in order to add a new list item to the urlpatterns list. This new item includes a path() that forwards requests with the pattern catalog/ to the module catalog.urls (the file with the relative URL **/catalog/urls.py**).

create a file inside your catalog folder called **urls.py**, and add the following text to define the (empty) imported urlpatterns. This is where we'll add our patterns as we build the application.

```
from django.conf.urls import url

from . import views


# Use include() to add paths from the catalog application
from django.urls import include

urlpatterns += [
    path('catalog/', include('catalog.urls')),
]


urlpatterns = [

]
```

Now let's redirect the root URL of our site (i.e. 127.0.0.1:8000) to the URL 127.0.0.1:8000/catalog/; this is the only app we'll be using in this project, so we might as well. To do this, we'll use a special view function (RedirectView), which takes as its first argument the new relative URL to redirect to (/catalog/) when the URL pattern specified in the path() function is matched (the root URL, in this case).

Add the following lines, again to the bottom of the file:

```
#Add URL maps to redirect the base URL to our application
from django.views.generic import RedirectView

urlpatterns += [
    path('/', RedirectView.as_view(url='/catalog/', permanent=True)),
]
```

Django does not serve static files like CSS, JavaScript, and images by default, but it can be useful for the development web server to do so while you're creating your site. As a final addition to this URL mapper, you can enable the serving of static files during development by appending the following lines.

Add the following final block to the bottom of the file now:

```
# Use static() to add url mapping to serve static files during development (only)
from django.conf import settings
from django.conf.urls.static import static

urlpatterns += static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

**Note**: There are a number of ways to extend the urlpatterns list (above we just appended a new list item using the += operator to clearly separate the old and new code). We could have instead just included this new pattern-map in the original list definition:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('catalog/', include('catalog.urls')),
    path('/', RedirectView.as_view(url='/catalog/', permanent=True)),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

In addition, we included the import line (from django.urls import include) with the code that uses it (so it is easy to see what we've added), but it is common to include all your import lines at the top of a Python file.

## Testing the website framework

At this point we have a complete skeleton project. The website doesn't actually *do* anything yet, but its worth running it to make sure that none of our changes have broken anything.

Before we do that, we should first run a *database migration*. This updates our database to include any models in our installed applications (and removes some build warnings).

## Running database migrations

Django uses an Object-Relational-Mapper (ORM) to map Model definitions in the Django code to the data structure used by the underlying database. As we change our model definitions, Django tracks the changes and can create database migration scripts (in **/library/catalog/migrations/**) to automatically migrate the underlying data structure in the database to match the model.

When we created the website Django automatically added a number of models for use by the admin section of the site (which we'll look at later). Run the following commands to define tables for those models in the database (make sure you are in the directory that contains **manage.py**):

```
python manage.py makemigrations
python manage.py migrate
```

**Important**: You'll need to run the above commands every time your models change in a way that will affect the structure of the data that needs to be stored (including both addition and removal of whole models and individual fields).

The makemigrations command *creates* (but does not apply) the migrations for all applications installed in your project (you can specify the application name as well to just run a migration for a single project). This gives you a chance to checkout the code for these migrations before they are applied — when you're a Django expert you may choose to tweak them slightly!

The migrate command actually applies the migrations to your database (Django tracks which ones have been added to the current database).

**Note**: See Migrations (Django docs) for additional information about the lesser-used migration commands.

## Running the website

During development you can test the website by first serving it using the *development web server*, and then viewing it on your local web browser.

**Note**: The development web server is not robust or performant enough for production use, but it is a very easy way to get your Django website up and running during development to give it a convenient quick test. By default it will serve the site to your local computer (http://127.0.0.1:8000/), but you can also specify other computers on your network to serve to. For more information see django-admin and manage.py: runserver (Django docs).

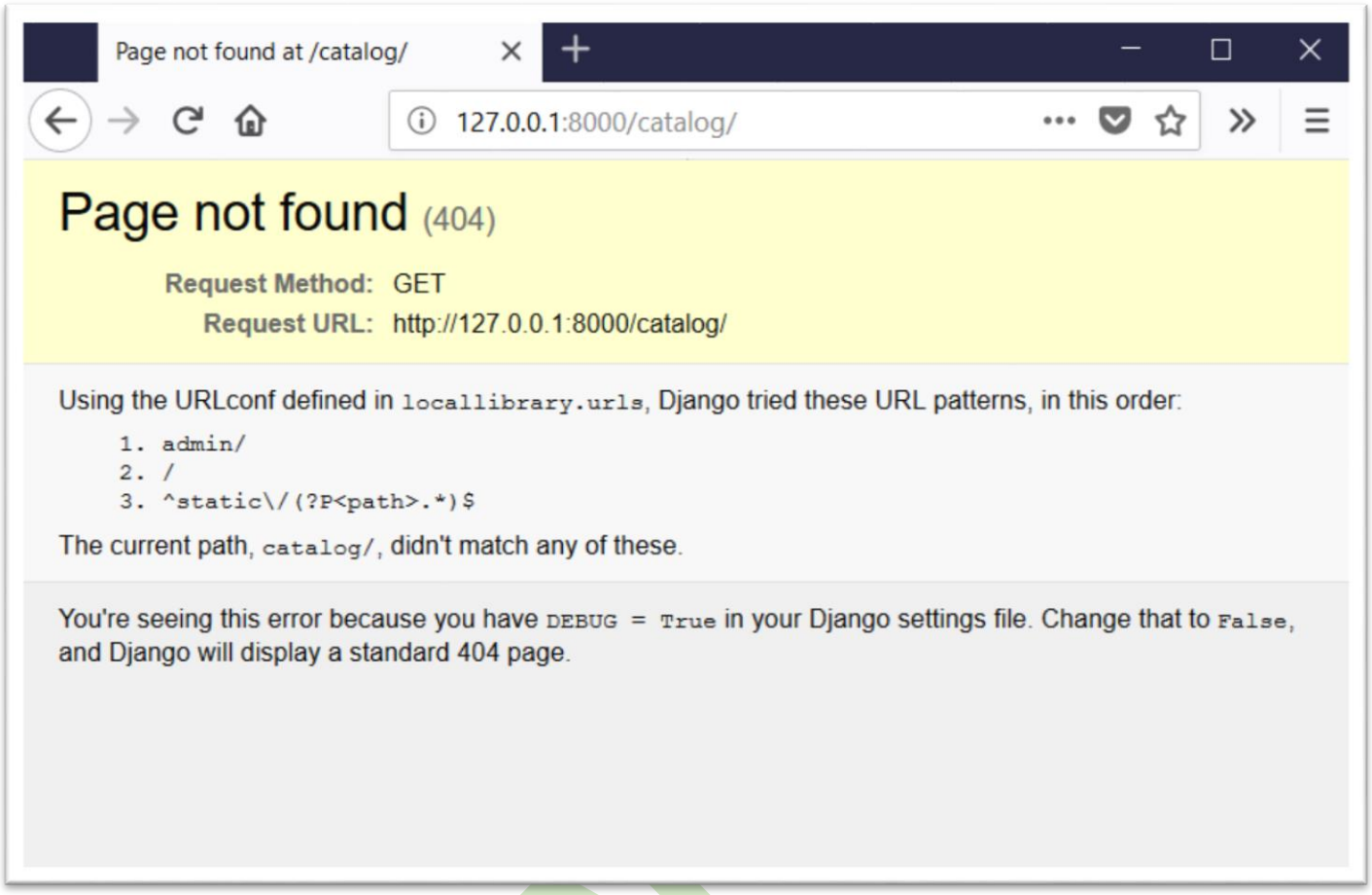
Run the *development web server* by calling the runserver command (in the same directory as **manage.py**):

```
python manage.py runserver
```

```
Performing system checks...

System check identified no issues (0 silenced).
September 22, 2016 - 16:11:26
Django version 1.10, using settings 'locallibrary.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Once the server is running you can view the site by navigating to http://127.0.0.1:8000/ in your local web browser. You should see a site error page that looks like this:




Don't worry! This error page is expected because we don't have any pages/urls defined in the catalogs.urls module (which we're redirected to when we get an URL to the root of the site).