Django Tutorial Part 2: Creating a skeleton website

Created by 이 윤 준 (yoonjoon.lee@gmail.com)

May, 2019

Overview

It is shown:

- how we can create a "skeleton" website,
- which we can then populate with site-specific settings, paths, models, views, and templates.

The process is straightforward:

1. Use the `django-admin` tool to create the project folder, basic file templates, and project management script (**manage.py**).
2. Use **manage.py** to create one or more *applications*.
3. Register the new applications to include them in the project.
4. Hook up the url/path mapper for each application.

For the Local Library website the website folder and its project folder will be named *locallibrary*, and we'll have just one application named *catalog*.

```
locallibrary/         # Website folder
    manage.py         # Script to run Django tools
                      # for this project (created using django-
    locallibrary/     # Website/project folder (created using d
    catalog/          # Application folder (created using manag
```

Creating the catalog application

First open a command prompt/terminal, make sure you are in your virtual environment, navigate to where you want to store your Django apps, and create a folder for your new website (in this case: *django_projects*). Then enter into the folder using the cd command:

```
mkdir django_projects
cd django_projects
```

Creating the project

Create the new project using the `django-admin startproject` command, and then navigate into the folder.

```
django-admin startproject locallibrary
cd locallibrary
```

The django-admin tool creates a folder/file structure.

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

Our current working directory should look something like this:

```
../django_projects/locallibrary/
```

The locallibrary project sub-folder is the entry point for the website:

- **__init__.py** is an empty file that instructs Python to treat this directory as a Python package.
- **settings.py** contains all the website settings. This is where we register any applications we create, the location of our static files, database configuration details, etc.
- **urls.py** defines the site url-to-view mappings. While this could contain all the url mapping code, it is more common to delegate some of the mapping to particular applications, as you'll see later.
- **wsgi.py** is used to help your Django application communicate with the web server. We can treat this as boilerplate.

The **manage.py** script is used to create applications, work with databases, and start the development web server.

Creating the catalog application

Run the following command to create the catalog application.

```
py -3 manage.py startapp catalog
```

The tool creates a new folder and populates it with files for the different parts of the application. 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:

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

- 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 recognise the folder as a Python Package and allow you to use its objects within other parts of the project.

Registering the catalog application

The application *catalog* has been created we have to register it with the project so that it will be included when any tools are run. Applications are registered by adding them to the `INSTALLED_APPS` list in the project settings.

Open the project settings file **django_projects/locallibrary/locallibrary/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`).

Specifying the database

The point where you would normally specify the database to be used for the project.

We'll use the SQLite database for this example and show how this database is configured in **settings.py**.

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

Other project settings

The **settings.py** file is also used for configuring a number of other settings.

We probably only want to change the `TIME_ZONE`.

```
TIME_ZONE = 'Asia/Seoul'
```

We 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, but for now we can keep it set to `True`.

Hooking up the URL mapper

The website is created with a URL mapper file (`urls.py`) in the project folder. It is more usual to defer mappings to the associated application.

```python
"""locallibrary URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/2.1/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),
]
```

Add a new list item to the urlpatterns list.

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

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

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/`. Add 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.

```
# 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)
```

As a final step, create a file inside your catalog folder called urls.py, and add the following text to define the (empty) imported urlpatterns.

```
from django.urls import path
from . import views

urlpatterns = [

]
```

Testing the website framework

At this point we have a complete skeleton project.

The website doesn't actually do anything yet, but it's worth running it to make sure that none of our changes have broken anything.

Running database migrations

As we change our model definitions, Django tracks the changes and can create database migration scripts (in **/locallibrary/catalog/migrations/**) to automatically migrate the underlying data structure in the database to match the model.

```
py -3 manage.py makemigrations
py -3 manage.py migrate
```

The `makemigrations` command creates the migrations for all applications installed in your project. The `migrate` command actually applies the migrations to your database.

Running the website

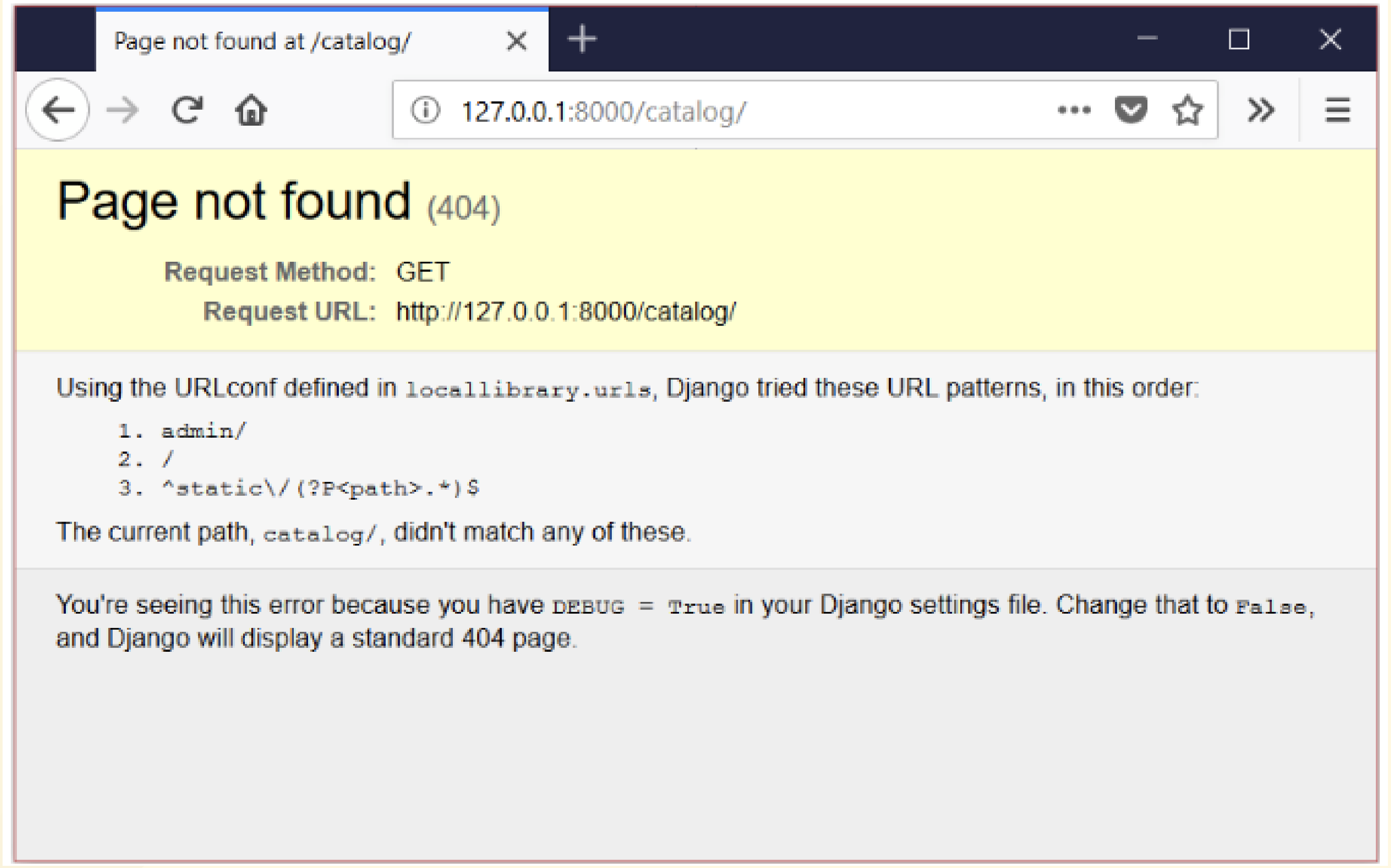
Run the development web server by calling the runserver command in the same directory as **manage.py**:

```
python3 manage.py runserver

 Performing system checks...

 System check identified no issues (0 silenced).
 August 15, 2018 - 16:11:26
 Django version 2.1, 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.



Page not found (404)

Request Method: GET
Request URL: http://127.0.0.1:8000/catalog/

Using the URLconf defined in `locallibrary.urls`, Django tried these URL patterns, in this order:

```
1. admin/
2. /
3. ^static\/(?P<path>.*)$
```

The current path, `catalog/`, didn't match any of these.

You're seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False`, and Django will display a standard 404 page.

Challenge yourself

A URL-mapping for the Admin site has already been added in the project's **urls.py**. Navigate to the admin area in your browser and see what happens.