NyanKiyoshi,
hello@vanille.bid
The 19th October, 2018.

Introduction Django Course

Table Of Contents

1 An Introduction to Django

Django is a productive way to write web applications using *Python*. It's a web framework here to help you build a software through a set of pre-made tools.

Among many other things, the *Django* framework comes bundled with a URL resolver, a template engine, a form builder, and an ORM engine as well.

2 Prerequisites for the course

- Some background knowledge in HTML and Python Object Programming ;
- Optionally create a *GitHub* account (free);
- A code editor (Sublime Text, Atom, vim, PyCharm, etc.) ;
- A terminal, whether it's on Windows, Linux or Mac OS ;
- Python > 3.4 (check by running `python3 --version`).

3 Installation of the other course requirements

3.1 Creating a Python Virtual Environment

After making sure you are all set to start, create a Python virtual environment by running `python3 -m venv myvirtualenv`.

Then, **you need to activate this environment** in your current terminal session, so the terminal is using your Python environment instead of the system python installation.

Unix systems (Mac OS included), `source myvirtualenv/bin/activate`

Windows, `myvirtualenv\Scripts\activate`

3.2 Installation of Git

Windows and Mac OS, install git-scm.com's software.

Ubuntu, run the command `apt install git`.

3.3 Installation of Django

In your activated Python environment, install *Django 2.1* using *pip* by running the following command: `pip install django~=2.1`.

4 Initialization of the project

Note: change the current directory to the target parent project directory.

4.1 Starting a new Django project

Create a new Django project by running the `django-admin` bootstrap, as follows:
`django-admin startproject` *`mysite`*

This will create the following file structure:

```
|-- manage.py
|-- mysite
    |-- settings.py
    |-- urls.py
    |-- wsgi.py
    |-- __init__.py

1 directory
5 regular files
```

`manage.py`	This file allows you to run commands on the project, like controlling the database or running the development HTTP server.
`mysite/settings.py`	The Django application settings file containing all your project configuration.
`mysite/urls.py`	A url mapping file to tell Django how to dispatch URLs. This file is also called the *URLconf file*
`mysite/wsgi.py`	The project WSGI application file, used to serve the django application with a production web server (nginx, Apache, etc.)
`mysite/__init__.py`	The standard Python package file.

4.2 Optional: adding the dependencies to a file

Using any editor, create a `requirements.txt` file containing the following content:

`requirements.txt`

```
django~=2.1

```

This file allows you to directly install the requirements by running:
`pip install -r requirements.txt` .

5 Configuration of the Django project

5.1 The basics

There are many available settings, and infinite ways to configure a base Django project. Some of them are:

`DEBUG`	A boolean that turns on/off the debug mode. Never deploy a site into production with `DEBUG` turned on.
`ALLOWED_HOSTS`	A list of strings representing the host/domain names that this Django site can serve. Example: `ALLOWED_HOSTS = ['localhost', '127.0.0.1']`
`LANGUAGE_CODE`	A string representing the language code for this installation, default is `'en-us'` for U.S. English.

You can find out more at this URL: `https://docs.djangoproject.com/en/2.1/ref/settings/`.

5.2 Setting up the database(s)

You can setup databases using the `DATABASES` settings, a dictionary containing the settings for all databases to be used with Django. It is a nested dictionary whose contents map a database alias to a dictionary containing the options for an individual database.

You must configure a `default` database among any (optional) additional databases. You can easily configure Django to use SQLite, MySQL, Postgres, etc.

More information at this URL: `https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-DATABASES`.

Example SQLite configuration:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mydatabase'
    }
```

```
}
```

Example Postgres configuration:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '5432'
    }
}
```

6 Starting up the development web server

Now that everything is set, you can start Django's development web server by running the following command: `python manage.py runserver` .

And then, if you open your web browser at `http://127.0.0.1:8000/`, you should see something like this:

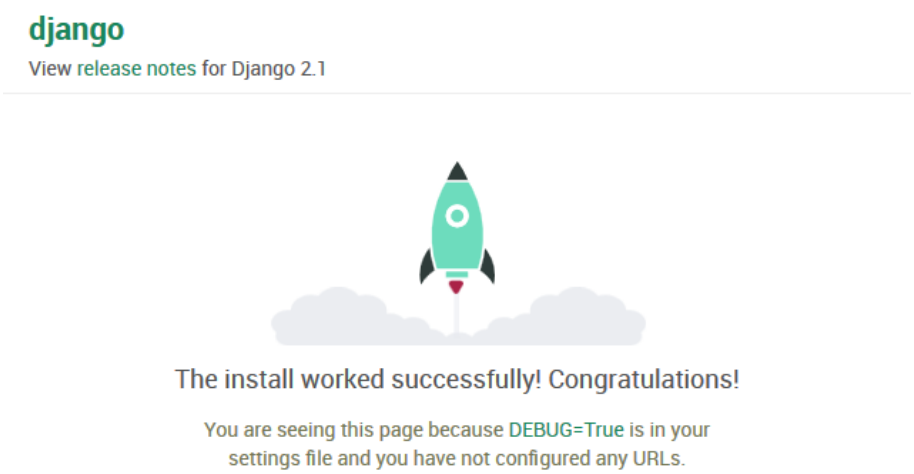


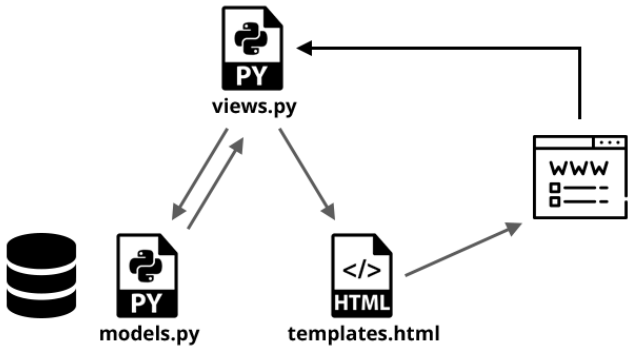
Figure 1: Default Django debugging index page.

You can also provide a host and a port, by doing like so for example: `python manage.py runserver 127.0.0.1:5000` .

Note: to stop the server, hit `Ctrl-C` in the terminal.

7 The design pattern of Django

Django uses a MTV (Model-Template-View) design pattern used to separate every part of your project into small applications.



Models: describes your **data** structure or database schema.

Views: controls **what** a user sees.

Templates: **how** a user sees it.

8 A demo project: a simple blog

To show how Django works, the best way is to look and learn from an example project code. So, here it is: let's build a simple blog. This blog will allow you to create and edit blog posts, and to post comments.

Run `python manage.py startapp blog` , to create our base blog application. Then, in your site settings (`mysite/settings.py`), add the `blog` application to the installed apps, by finding the `INSTALLED_APPS` variable and appending `blog` to it. Like so:

mysite/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog' # our blog application added here
]
```

8.1 The Models

Let's create a blog post model for our database and us to use later in our application to create, manage and view our blog and its posts.

blog/models.py

```
from django.db import models
from django.utils import timezone


class Post(models.Model): # This line defines our Django ORM model.
    author = models.ForeignKey('auth.User', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(default=timezone.now)

    def __str__(self):
        return self.title

```

Let's get line by line what that block of code does. First, we have:

- `class Post(models.Model):` this line defines our Django ORM model, where `models.Model` is the base Django ORM model and `Post`, the name of our model.
- Then, we defined some properties: `author`, `title`, `text` and `created_date`, with a given type (like a relation, a text, a number, and a date).

  We have:

  - `models.ForeignKey` this defines a relation (or a link) to another model.
  - `models.CharField` this defines a short text field, it has a limited length.
  - `models.TextField` this defines a long text field (ideal for a blog article's content).
  - `models.DateTimeField` this defines a date and time object.

- And finally, we have `def __str__(self):` that allows us to get the blog title when we want to show the blog post entry, that will be the `Post` object.

8.1.1 Creating the migrations and tables

Now that we have the models, we need to create the django migrations and then SQL tables through the migrations.

To generate the models migrations, run `python manage.py makemigrations blog` . This should output the following:

```
Migrations for 'blog':
  blog/migrations/0001_initial.py:
  - Create model Post
```

Then, we can run `python manage.py migrate blog` to create the database tables, and we should get the following output:

```
Operations to perform:
  Apply all migrations: blog
Running migrations:
  Applying blog.0001_initial... OK
```

8.2 The Admin Site

Now, we can use one of the features of the Django framework to quickly give us a way to manage our newly created model (retrieve, create, update, delete).

For this, add the following content to the file `blog/admin.py` :

```python
from django.contrib import admin
from .models import Post

admin.site.register(Post)

```

This will register our `Post` model to the admin page. You can now open your web browser to `http://127.0.0.1:8000/admin/`, and you should see the following page:

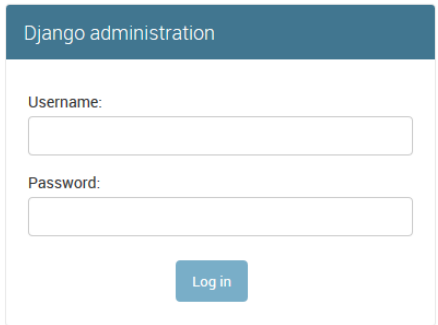


Figure 2: The django-admin login page.

8.2.1 Creating the admin credentials

To login, we need to create a new admin user (a super-user). To do that, we need to run the following command: `python manage.py createsuperuser` and fill everything. You will get something like this:

```
Username (leave blank to use 'myusername'): admin
Email address: admin@example.com
Password:
Password (again):
The password is too similar to the email address.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Then we should be able to login if you return on the admin page. And then we should see this:

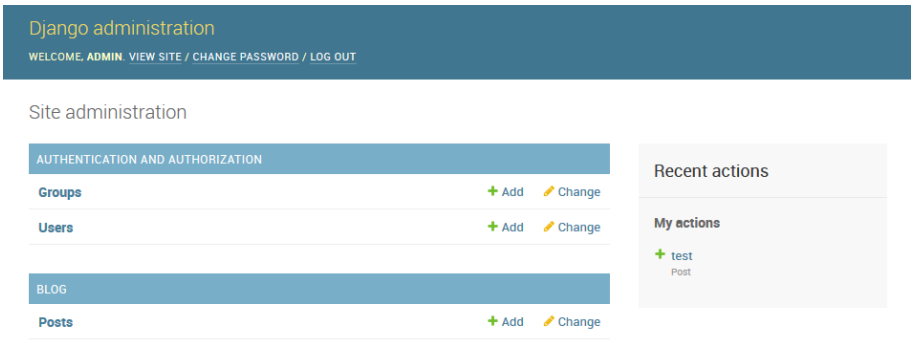


Figure 3: Our django-admin index page, with the blog post model.

You can now play around with the posts, like adding a few posts, editing them, deleting them, etc.

8.3 The Views

Let's put some logic to retrieve article and show our posts on the homepage.

blog/views.py

```
from django.shortcuts import render
from .models import Post


def post_list(request):
    context = {'posts': Post.objects.all()}
    return render(request, 'blog/post_list.html', context)

```

- First, we create a `post_list` view that we will later route to a URL;
- Then, we create a context (a dictionary) for the template to be rendered;
- We store as `'post'` all the existing posts in the database by using the `Post.objects.all()` instruction, provided by Django ORM;
- Then, we tell Django to render our (non-existing) template `blog/post_list.html` and we pass our context containing the blog posts.

8.4 The URL configuration

Now that we have our view, we can route it. For that, create a URLConf file for our `blog` package (`blog/urls.py`), containing:

blog/urls.py

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

urlpatterns = [
    path('', views.post_list, name='post-list')
]

```

- We create a list of URL patterns;
- We route as root point our `views.post_list` view and we tell Django that we want to internally call it `post-list` . The name will only be

visible for the developer, and not for the user, it will allow us later to tell Django that we are speaking about that URL and not another one.

8.5 Telling Django about our package URLs

What we have done is not enough, if you test, Django doesn't detect our new URL. And that is because we need to tell our site (or our core) to include the URLs of our `blog` package.

For that, we edit `mysite/urls.py` to:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path, include

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

    # include the URLs of the 'blog' package
    path('', include('blog.urls'))
]

```

If you look closely, you will notice a new instruction: `path('', include('blog.urls'))`. This instructs Django to route all of our blog URLs to the root point (this will be `http://127.0.0.1:8000/`).

That's as simple as that.

8.6 The Templates

Now, if you open your web browser back to `http://127.0.0.1:8000/` you should see and notice this error:

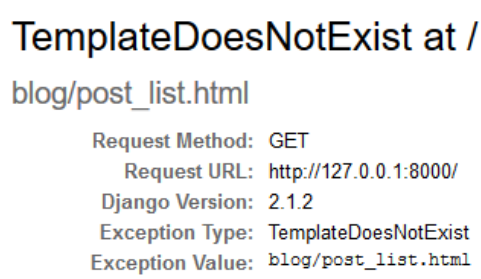


Figure 4: Django failed loading a template.

That is because we have yet to create the `blog/post_list` template that we used in our view.

8.6.1 The First Template of Our Project

We create a `templates` directory into the `blog` package, then for better code understanding, a `blog` directory inside it (`templates/blog`).

Now, we create the template file:

blog/templates/blog/post_list.html

```
{% for post in posts %}
  <div>
    <p>Date: {{ post.created_date }}</p>
    <h2>
      <a href="#">{{ post }}</a>
    </h2>
    <p>{{ post.text | linebreaksbr }}</p>
  </div>
{% endfor %}

```

8.6.2 Template Inheritance

8.7 Adding some CSS and extending the templates

Now, let's make our blog look better. For that, we are going to use the bootstrap framework.

8.8 Installing the bootstrap framework

We need to transform our `post_list.html` to a valid HTML template as done below. And then, we have to install Bootstrap 4 in the `head` block of our HTML code.

blog/templates/post_list.html

```
<!doctype HTML>
<html>
  <head>
<link
rel="stylesheet"
href="//stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
/>
  </head>
  <body>
    <div class="container">
      {% for post in posts %}
        <div>
          <h2>
            <a href="#">{{ post }}</a>
```

```
          </h2>
          <p class="text-muted">Date: {{ post.created_date }}</p>
          <p>{{ post.text | linebreaksbr }}</p>
        </div>
      {% endfor %}
    </div>
  </body>
</html>

```

8.9 Extending the templates

Later in our blog, we will have multiple and different templates and pages: one to list the article, one to view a single article. One issue we will face with that is that we will have to repeat some base structure code of our HTML template.

The solution for that is extending a given base template and only adding whatever new content we want.

8.9.1 Creating the base template

Let's create a `base.html` file that will contain the base HTML structure of `post_list.html`.

blog/templates/base.html

```
<!doctype HTML>
<html>
  <head>
<link
rel="stylesheet"
href="//stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
/>
  </head>
  <body>
    <div class="container">
      {% block content %}{% endblock %}
    </div>
  </body>
</html>

```

As you may note here, we removed the `for` loop that we had, so we removed everything that is used to show the post list. Instead, we replaced the content with `{% block content %}{% endblock %}`.

The whole code of `base.html` will be extended from other templates (where `post_list.html` is one of them). Then, we have the content block (

`{% block content %}{% endblock %}`) that will be overridden by the templates extending it to add their own content to the base template.

8.9.2 Updating the post listing template

Now, we need to extend `base.html` and override the `content` block.

blog/templates/post_list.html

```
{% extends 'blog/base.html' %}
{% block content %}
  {% for post in posts %}
    <div>
      <h2>
        <a href="#">{{ post }}</a>
      </h2>
      <p class="text-muted">Date: {{ post.created_date }}</p>
      <p>{{ post.text | linebreaksbr }}</p>
    </div>
  {% endfor %}
{% endblock %}

```

8.10 Viewing comments over a single blog post

For the *section 8.11* demo example, we woud like to prepare a little 'comments' feature into our blog. We are gonna add a new `Comment` model and clicking on the posts title will now redirect the user to the single view by updating the template, URLs and views.

8.10.1 Adding the new models

8.10.2 Updating the URLs and views

8.10.3 Updating the templates

8.11 The Forms

8.11.1 CSRF

+ custom Field for calculating am I a bot?

9 What's next?

Any questions, troubles or suggestion?

GitHub

Fork me or open an issue on the GitHub repository!

hello@vanille.bid
https://github.com/VanilleBid/django-stream-2018