

NYANKIYOSHI,
HELLO@VANILLE.BID
THE 19TH OCTOBER, 2018.

Introduction Django Course

Table Of Contents

1	An Introduction to Django	3
2	Prerequisites for the course	3
3	Installation of the other course requirements	3
	3.1 Creating a Python Virtual Environment	3
	3.2 Installation of Git	3
	3.3 Installation of Django	3
4	Initialization of the project	4
	4.1 Starting a new Django project	4
	4.2 Optional: adding the dependencies to a file	4
5	Configuration of the Django project	5
	5.1 The basics	5
	5.2 Setting up the database(s)	5
6	Starting up the development web server	6
7	The design pattern of Django	7
8	A demo project: a simple blog	7
	8.1 The Models	8
	8.1.1 Creating the migrations and tables	9
	8.2 The Admin Site	9
	8.2.1 Creating the admin credentials	10
	8.3 The Views	11
	8.4 The URL configuration	11
	8.5 Telling Django about our package URLs	12
	8.6 The Templates	12

	8.6.1	The First Template of Our Project	13
	8.6.2	Template Inheritance	13
8.7		Adding some CSS and extending the templates	13
8.8		Installing the bootstrap framework	13
8.9		Extending the templates	14
	8.9.1	Creating the base template	14
	8.9.2	Updating the post listing template	15
8.10		Viewing comments over a single blog post	15
	8.10.1	Adding the new models	15
	8.10.2	Updating the views	16
	8.10.3	Updating the URLs	17
	8.10.4	Updating the templates	18
8.11		The Forms	18
8.12		The ModelForm	19
	8.12.1	Integrating the form to the views	19
	8.12.2	Updating the templates	20
	8.12.3	The CSRF protection	22
9		What's next?	23

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, HTTP protocol 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),	<code>source myvirtualenv/bin/activate</code>
Windows,	<code>myvirtualenv\Scripts\activate</code>

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:

```
1 |-- manage.py
2 |-- mysite
3   |-- settings.py
4   |-- urls.py
5   |-- wsgi.py
6   |-- __init__.py
7
8 1 directory
9 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`

```
1 django~=2.1
2
```

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`

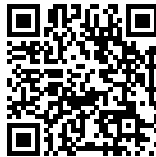
A list of strings representing the host/domain names that this Django site can serve.

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:

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         'NAME': 'mydatabase'  
5     }  
}
```

```
6 }
```

Example Postgres configuration:

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.postgresql',  
4         'NAME': 'mydatabase',  
5         'USER': 'mydatabaseuser',  
6         'PASSWORD': 'mypassword',  
7         'HOST': '127.0.0.1',  
8         'PORT': '5432'  
9     }  
10 }
```

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:

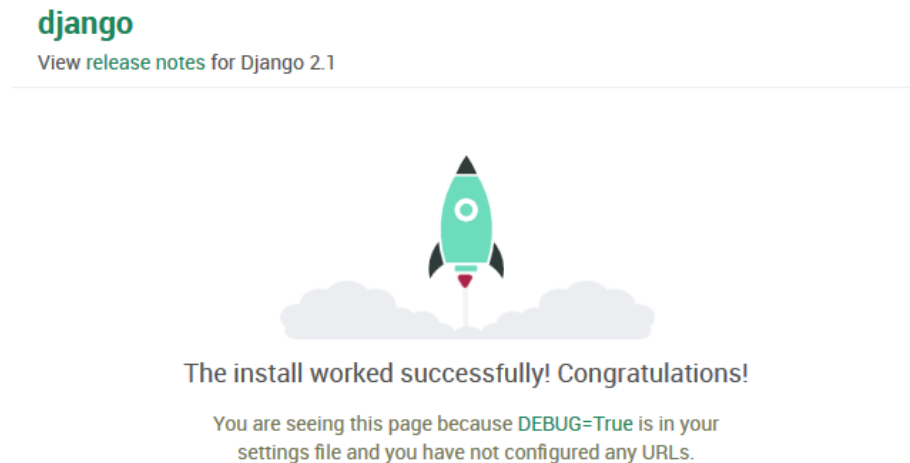


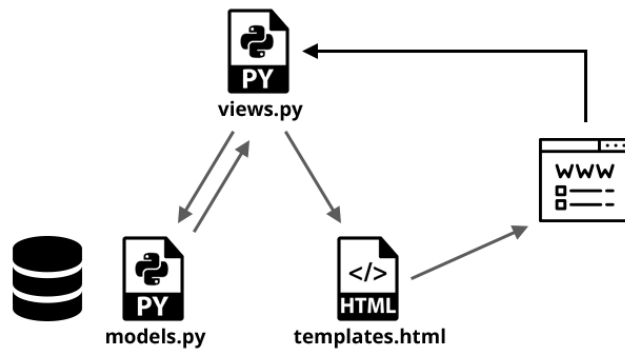
Figure 1: Default Django debugging index page.

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 use it later in our application to create, manage and view our blog and its posts.

```

                                blog/models.py
1 from django.db import models
2 from django.utils import timezone
3
4
5 class Post(models.Model): # This line defines our Django ORM model.
6     author = models.ForeignKey('auth.User', on_delete=models.CASCADE)
7     title = models.CharField(max_length=200)
8     text = models.TextField()
9     created_date = models.DateTimeField(default=timezone.now)
10
11     def __str__(self):
12         return self.title
13

```

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
```

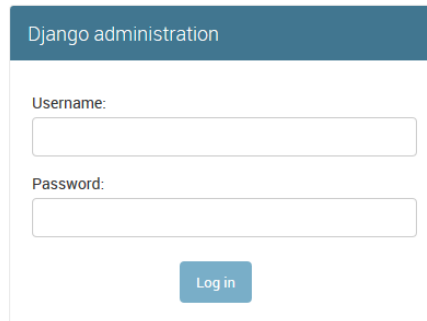
8.2 The Admin Site

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`:

```
1 from django.contrib import admin
2 from .models import Post
3
4 admin.site.register(Post)
5
```

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:



The image shows the Django administration login page. It has a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". Below the "Password:" field, there is a blue button labeled "Log in".

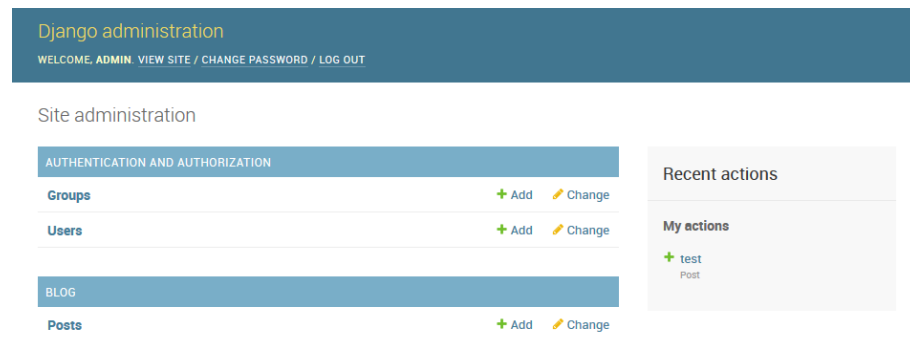
Figure 2: The django-admin login page.

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:



The image shows the Django administration index page. At the top, there is a dark blue header with the text "Django administration" and "WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below the header, there is a section titled "Site administration". This section contains two main categories: "AUTHENTICATION AND AUTHORIZATION" and "BLOG". Under "AUTHENTICATION AND AUTHORIZATION", there are two sub-sections: "Groups" and "Users". Each sub-section has a "+ Add" button and a "Change" button. Under "BLOG", there is a sub-section: "Posts", which also has a "+ Add" button and a "Change" button. To the right of the "Site administration" section, there is a "Recent actions" section. This section contains a "My actions" sub-section, which has a "+ test" button and a "Post" button.

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

```
1 from django.shortcuts import render
2 from .models import Post
3
4
5 def post_list(request):
6     context = {'posts': Post.objects.all()}
7     return render(request, 'blog/post_list.html', context)
8
```

- 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

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.post_list, name='post-list')
6 ]
7
```

- 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
1 from django.contrib import admin
2 from django.urls import path, include
3
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6
7     # include the URLs of the 'blog' package
8     path('', include('blog.urls'))
9 ]
10
```

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:

```
TemplateDoesNotExist at /
blog/post_list.html
Request Method: GET
Request URL: http://127.0.0.1:8000/
Django Version: 2.1.2
Exception Type: TemplateDoesNotExist
Exception Value: blog/post_list.html
```

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

```
1 {% for post in posts %}
2   <div>
3     <p>Date: {{ post.created_date }}</p>
4     <h2>
5       <a href="#">{{ post }}</a>
6     </h2>
7     <p>{{ post.text | linebreaksbr }}</p>
8   </div>
9 {% 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

```
1 <!doctype HTML>
2 <html>
3   <head>
4     <link
5       rel="stylesheet"
6       href="//stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
7     />
8   </head>
9   <body>
10     <div class="container">
11       {% for post in posts %}
12         <div>
13           <h2>
14             <a href="#">{{ post }}</a>
```

```

15         </h2>
16         <p class="text-muted">Date: {{ post.created_date }}</p>
17         <p>{{ post.text | linebreaksbr }}</p>
18     </div>
19     {% endfor %}
20 </div>
21 </body>
22 </html>
23

```

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

```

1 <!doctype HTML>
2 <html>
3   <head>
4     <link
5       rel="stylesheet"
6       href="//stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
7     />
8   </head>
9   <body>
10     <div class="container">
11       <div class="navbar">
12         <span class="navbar-brand">
13           <a href="{% url 'post-list' %}">Blog Demo</a>
14         </span>
15       </div>
16       {% block content %}{% endblock %}
17     </div>
18   </body>
19 </html>
20

```

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

```
1 {% extends 'blog/base.html' %}
2 {% block content %}
3     {% for post in posts %}
4         <div>
5             <h2>
6                 <a href="#">{{ post }}</a>
7             </h2>
8             <p class="text-muted">Date: {{ post.created_date }}</p>
9             <p>{{ post.text | linebreaksbr }}</p>
10        </div>
11    {% endfor %}
12 {% endblock %}
13
```

8.10 Viewing comments over a single blog post

For the *section 8.11* demo example, we would like to prepare a little comments section 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

Let's create a `Comment` model with a name, a content and a `created_date` field. We will append it to `models.py` .

blog/models.py

```
1 class Comment(models.Model):
2     post = models.ForeignKey(
3         to=Post, related_name='comments', null=False, blank=False,
4         on_delete=models.CASCADE)
5     author_name = models.CharField(max_length=40)
6     content = models.CharField(max_length=500)
7
```

```

8 class Meta:
9     ordering = ('-pk',)
10

```

Little notes on how the `ForeignKey` field works:

`on_delete`, this is the behaviour to adopt when the referenced object is deleted.

- `models.CASCADE` means that when the referenced object is deleted, it must also delete the objects that have references to it.
- `models.PROTECTED` it forbids the deletion of the referenced object until the references are not removed.
- `models.SET_NULL` means to set the referenced objects to `NULL`.

Now, generate the new migrations file by running like before:

`python manage.py makemigrations blog` which should output:

```

Migrations for 'blog':
  blog\migrations\0002_auto_20181023_2205.py
    - Create model Comment
    - Add field comments to post

```

Then, apply the migrations by running

`python manage.py migrate blog`. Which, should output:

```

Operations to perform:
  Apply all migrations: blog
Running migrations:
  Applying blog.0002_auto_20181023_2205... OK

```

8.10.2 Updating the views

First, we want to add a view that will allow us to view a single post entry. For that, we will update the `views.py` file with this content:

blog/views.py

```

1 from django.http import Http404
2 from django.shortcuts import render
3 from .models import Post
4
5
6 def post_list(request):
7     context = {'posts': Post.objects.all()}
8     return render(request, 'blog/post_list.html', context)
9
10
11 def post_single(request, post_id: int):

```



```

12     try:
13         # Prefetch the comments and retrieve the given post
14         post = Post.objects.prefetch_related('comments').get(pk=post_id)
15     except Post.DoesNotExist:
16         raise Http404(f'Post #{post_id} not found.')
17
18     return render(request, 'blog/post_single.html', {'post': post})
19

```

First of all (at line 1), we imported `Http404` from `django.http` to allow us to throw HTTP 404 errors on demand.

Then, at line 11, we define a new view called `post_single` taking a `post_id` parameter of the type `int`, it will allow us to pass the `post_id` parameter from the URL.

After that we tell the query set of the Django's ORM that it will have to prefetch the `Post` comments. This will allow Django to know in advance that we will need the comments and allow us to make the SQL request in one go, instead of doing one at the beginning (here) and again later in the templates code when we will want the comments. So, for example, locally, it saves 1ms for the response time.

Then, we say that we want to retrieve the post having as primary key the given post identifier.

But, if the post does not exist (e.g. the user wanted the post number 99), it will raise the `Post.DoesNotExist` exception that we handle to raise a HTTP 404 error instead.

Then, at line 18, we render the single post template with the post.

8.10.3 Updating the URLs

Now, we update `urls.py` to route a single post to our newly created `post_single` view.

blog/urls.py

```

1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.post_list, name='post-list'),
6     path('post/<int:post_id>', views.post_single, name='post-single')
7 ]
8

```

You may notice at line 6, that you have `post/<int:post_id>`, this tells Django that we are expecting an integer parameter in the URL and that we want it to be passed to the view as `post_id`.

8.10.4 Updating the templates

Now we can route our posts to their single post view, to do that, we update the link in `post_list.html` to this:

```
<a href="{% url 'post-single' post_id=post.pk %}">{{ post }}</a>
```

This will tell Django to generate a URL to a single post with the identifier of the currently proceeded post.

Then, we want to create the template `post_single.html` to show a single post and show the comments. We will put that:

blog/templates/post_single.html

```
1 {% extends 'blog/base.html' %}
2
3 {% block content %}
4     <div class="card">
5         <div class="card-body">
6             <h5 class="card-title">{{ post.title }}</h5>
7             <p class="card-text">{{ post.content | linebreaksbr }}</p>
8         </div>
9     </div>
10
11     {% with comments=post.comments.all %}
12     {% if comments %}
13         <div class="card mt-4" id="comments">
14             <div class="card-body">
15                 <h5 class="card-title">Comments</h5>
16                 <ul class="list-group list-group-flush">
17                     {% for comment in comments %}
18                     <li class="list-group-item">
19                         {{ comment.name }} {{ comment.content }}
20                     </li>
21                     {% endfor %}
22                 </ul>
23             </div>
24         </div>
25     {% endif %}
26     {% endwith %}
27 {% endblock %}
28
```

8.11 The Forms

Now that we have a page for viewing a single post, and comments models and templates up and running, we can create a form allowing the user to add comments to our posts.

8.12 The ModelForm

In a new file named `forms.py`, we are going to extend a base form class that we will use as our form builder. This base form builder class is called `ModelForm` because it takes a `model` *meta attribute*¹ telling the form builder around what type of data model to build.

blog/forms.py

```
1 from django import forms
2 from .models import Comment
3
4
5 class CommentForm(forms.ModelForm):
6     class Meta:
7         model = Comment
8         fields = ('author_name', 'content')
9
```

At line 7, we are telling that we want to use the `Comment` model for the form.

At line 8, we then tell we only want to use and only handle the `author_name` and `content` field of `Comment`, thus, we exclude any field that we don't want the user to be able to manipulate or inject.

That's it.

8.12.1 Integrating the form to the views

We need to update the code base of the `post_single` view, to integrate the `CommentForm` form. Here is how we are going to proceed:

First, we construct, we need to initialize `CommentForm` with:

- The HTTP POST data or nothing (`None`) is there is no data;
- A target `Comment` instance where the form will commit the changes.

¹ To learn about meta classes, you can take a look at <https://jfine-python-classes.readthedocs.io/en/latest/metaclass-attribute.html>



Then, we check, we check if the form was submitted and is valid, using the `is_valid` method.

We commit, and redirect the user if the form is valid;

We render the form, if the form is not valid.

blog/views.py

```
def post_single(request, post_id: int):
    try:
        # Prefetch the comments and retrieve the given post
        post = Post.objects.prefetch_related('comments').get(pk=post_id)
    except Post.DoesNotExist:
        raise Http404(f'Post #{post_id} not found.')

    # create a base empty comment to put user input into;
    # this base comment will have has parent post the current viewing
    # post.
    comment_instance = Comment(post=post)

    # we create the form from the HTTP POST data or pass None if no POST
    # data;
    # and we pass our empty comment instance to be populated.
    comment_form = CommentForm(request.POST or None,
                               instance=comment_instance)

    # check if the form is valid
    if comment_form.is_valid():
        # commit the changes if the form is valid
        comment_form.save(True)

        # and redirect the user back to the single post view
        return redirect('post-single', post_id=post_id)

    # pass the form to render
    context = {'post': post, 'comment_form': comment_form}
    return render(request, 'blog/post_single.html', context)
```

8.12.2 Updating the templates

Now, we just have to render the form as a HTML form in the single post template. For that, we just have to write `{{ form }}` in the template, it will render its `<label>` and `<input>` automatically (inline form). Or call `form.as_p` to render the form wrapped around `<p>` tags (vertical form).

Note: there is also `form.as_table` to wrap around `<tr>` tags and `form.as_ul` to wrap around `` tags.

Here is the final code (to add after the `{%{ endwhile %}}` instruction):

blog/templates/post_single.html

```
<form method="post">
  {{ comment_form.as_p }}
  <button class="btn btn-primary" type="submit">Submit</button>
</form>
```

Note: the code doesn't work as is, see the next section.

8.12.3 The CSRF protection

You may notice that if you run the above code as is, you run into this error:

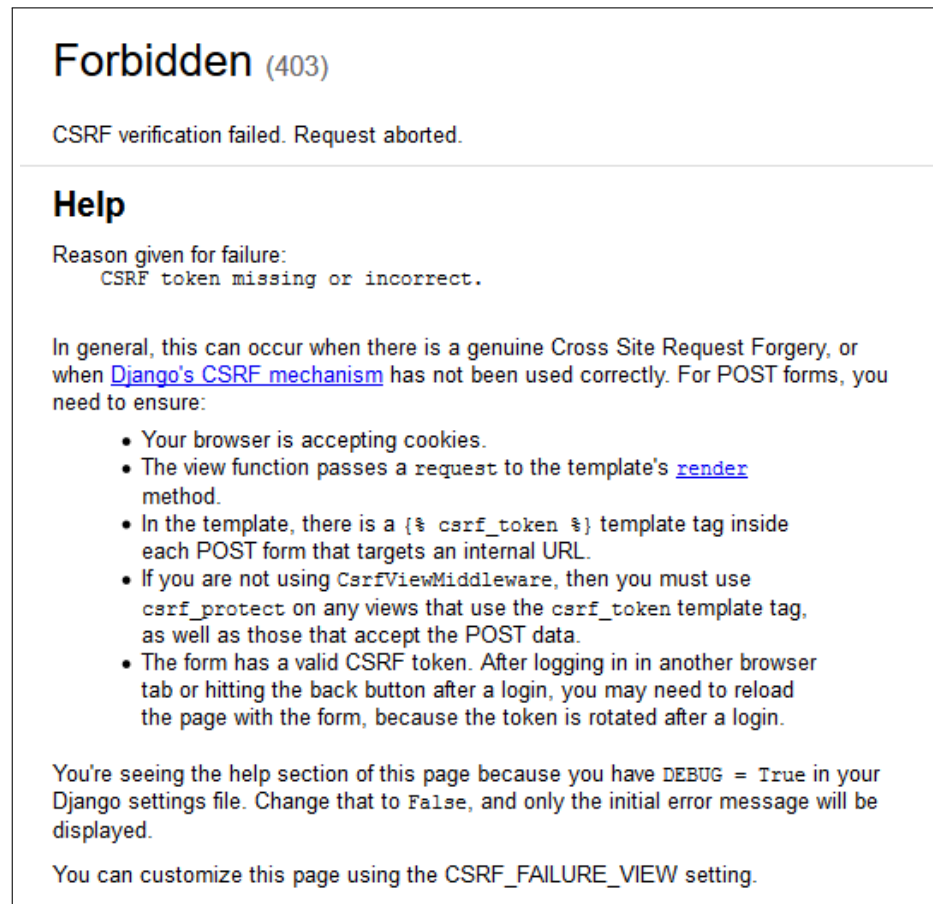


Figure 5: CSRF verification failure error with `DEBUG=True`.

The reason behind this error, is that by default, Django is enabling the CSRF protection which is here to defend your website against the Cross-site Scripting (XSS) vulnerability. To be short: CSRF is an attack where a malicious entity manage to tricks a victim into performing actions on behalf of the attacker (e.g.: clicking on a link)².

And one of the most popular protection against this type of attack, is to require the user to give back a randomly generated token that was given to the browser when getting the form.

² More information at: <https://www.acunetix.com/websecurity/csrf-attacks/>

And, in our above form, we did not give the user such CSRF token. To fix that, we need to call the tag `csrf_token` in our `<form>`'s body. This will generate a hidden `<input>` field containing a CSRF token, in addition to a cookie.

So, we have to edit our code to this:

blog/templates/post_single.html

```
<form method="post">
  {% csrf_token %} <!-- instruction to generate a CSRF token -->
  {{ comment_form.as_p }}
  <button class="btn btn-primary" type="submit">Submit</button>
</form>
```

9 What's next?

- What about making a blog dashboard without using `django-admin` (`admin.py`) by only using django forms, views and templates?
- What about a todo list application?

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>