TASK

Django – Sticky Notes Application Part 1

Visit our website

Introduction

In the previous tasks, we have been incrementally introducing you to the skills you need to create an object-oriented software application. You learnt about software design and tools you can use to simplify and manage complex processes. Then, you learnt about unit testing, setting up a professional grade project with virtual environments and PEP-compliant code, and the implementation of modular designs.

Ready to build your first Django app? In this task, you will become familiar with Django and create a sticky notes application from scratch. We will cover essential concepts like URL routing, template creation, CSS styling, static file management, and database migrations. By the end, you will have a fully functional app that showcases your newfound Django skills and demonstrates your understanding of software development best practices. Let's get started!

DJANGO

In this task, we will focus on introducing you to the Django framework. It's a versatile open-source web framework written in Python that empowers you to swiftly develop secure and scalable websites and web applications. Renowned platforms like Instagram and Spotify have embraced Django for its emphasis on efficient code reuse and rapid implementation, guided by design principles such as "don't repeat yourself" (DRY). By leveraging Django's powerful libraries and tools, including object-relational mapping and optional administrative interfaces, you will gain the ability to construct robust, data-driven apps with code that is not only faster to implement, but also cleaner and more pragmatic than alternative frameworks.

OBJECT-RELATIONAL MAPPING

In modelling business applications, two conflicting paradigms often come into play: the object-oriented model and the relational model.

The object-oriented approach is based on exposing behaviour, hiding data, and leveraging code reuse through inheritance, prioritising behavioural modelling over data-centric considerations. This sharply contrasts with the relational model, which lacks a concept of behaviour and is centred on relationships between data entities, emphasising data modelling over behaviour.

The challenge arises when attempting to seamlessly translate designs between these two modelling techniques. One notable instance of this conflict arises when dealing with attributes in a class that are represented as lists or dictionaries. In the relational model, there is no designated data type for collection fields. Instead, one must create a table to store the data present in the list or dictionary and establish relationships using primary keys referencing the object.

Consequently, when designing applications, a conversion becomes imperative to harness the modularity offered by object-oriented programming and the data integrity inherent in the relational model. Many enterprise applications address this challenge automatically by employing object-relational mappers (ORMs). Understanding the purpose of these mappers is crucial, as they enable developers to seamlessly integrate the benefits of both paradigms, enhancing productivity and maintainability in the process.

Django is known for being a rapid application development (RAD) framework because it enables you to build applications without worrying about mundane details such as the generated SQL queries for your application. That is partly due to Django's built-in ORM, which you will learn about later in this lesson when you begin working with Django models.

MVT ARCHITECTURE

The **Model-View-Template (MVT)** architecture is a design pattern associated with Django. MVT is Django's variation of the more common **Model-View-Controller (MVC)** architecture. In Django's MVT, the components are **Model**, **View**, and **Template**, each with specific responsibilities.

Model (M)

The **Model** component represents the business logic and data structure of the application. Business logic refers to the rules, processes, and calculations that dictate how an application operates and how data is processed. It encapsulates the specific operations and workflows that are unique to a particular business or application domain. A data structure is a way of organising and storing data to facilitate efficient manipulation and retrieval, and defining the format in which data is stored and how relationships between data elements are managed.

The Model component’s responsibilities are to:

- represent the structure of the application's data,
- define models as Python classes, each representing a table in the database, and
- use Django's ORM to interact with the database, abstracting SQL queries.

In a Django app, **models.py** contains the model classes. Migrations are used to synchronise the database with model changes.

View (V)

The **View** component handles the interaction between the user and the application, managing the presentation logic.

The View component’s responsibilities are to:

- handle user requests and define the logic for processing them,
- interact with models to retrieve or update data, and
- return appropriate HTTP responses, such as rendering templates or redirecting.

In a Django app, **views.py** contains view functions or classes. Views define the behaviour of different URL patterns.

HTTP request handling (Controller)

Usually, the **Controller** manages the flow of data between the Model and the View. Controllers are not explicitly defined in Django, as its MVT pattern incorporates Controller-like functionality within the framework.

The traditional responsibility of the Controller is to handle HTTP request processing and routing which is implicitly handled by Django's framework. The Views component in Django often includes the logic traditionally associated with Controllers.

Template (T)

The **Template** component deals with the presentation layer, defining the structure and appearance of the HTML content.

The Template component's responsibilities are to:

- define the structure of the HTML pages,
- incorporate dynamic data using template tags, and
- receive data from views through context dictionaries.

In a Django app, HTML templates are stored in the **templates** directory. HTML pages are constructed using template tags for data integration.

Interactions between components

Model-View interaction:

- The View interacts with the Model to retrieve data needed for presentation.
- The View can modify data in the Model based on user interactions.

View-Template interaction:

- The View updates the Template with data from the Model.
- The Template is responsible for displaying the data in a user-friendly format.

Template-Model interaction:

- The Template can update the Model indirectly through user interactions.
- The Template reflects changes in the Model by dynamically updating the HTML content.

Distinguishing MVT from MVC

- Django's MVT is a variation of the traditional MVC pattern.
- In MVT, the Template serves as a dedicated layer for handling presentation concerns, whereas MVC treats the View as the Template. Also, the View in Django's MVT takes the role of the Controller in MVC.

`HTTPRequest` class:

The `HTTPRequest` class represents an incoming HTTP request and provides access to a variety of information about the request, including `GET` and `POST` parameters.

Structure of the `HTTPRequest` class:

Attributes:

- `method`: HTTP method used for the request (e.g., `GET`, `POST`).
- `GET`: A dictionary containing `GET` parameters.
- `POST`: A dictionary containing `POST` parameters.
- `COOKIES`: A dictionary containing cookies sent with the request.
- `META`: A dictionary containing additional HTTP headers.

Methods:

- `get_full_path()`: Returns the path, plus an appended query string, if applicable.
- `is_secure()`: Returns `True` if the request was made over HTTPS.
- `build_absolute_uri(location)`: Returns an absolute URI based on the location argument.

Accessing `GET` parameters:

- `request.GET`: Dictionary-like object containing all `GET` parameters.
- `request.GET.get('parameter_name')`: Get the value of a specific `GET` parameter.

Accessing `POST` parameters:

- `request.POST`: Dictionary-like object containing all `POST` parameters.
- `request.POST.get('parameter_name')`: Get the value of a specific `POST` parameter.

An example of how this could be structured is as follows:

```
from django.http import HttpRequest

def my_view(request: HttpRequest):
    # Accessing GET parameters
    param_value = request.GET.get('param_name', default_value)

    # Accessing POST parameters
    post_param_value = request.POST.get('post_param_name',
default_value)

    # Other attributes and methods
    method = request.method
    is_https = request.is_secure()
    full_path = request.get_full_path()

    # ...

    return HttpResponse("Response content")
```

Django's MVT architecture provides a structured and efficient way to develop web applications by separating concerns and promoting modularity. This design pattern enhances maintainability, scalability, and code readability in Django projects.

BUILD A DJANGO APPLICATION

In the next part of this lesson, our attention shifts towards hands-on application as we begin building a Django bulletin board application. This example bulletin board project serves as a foundational step, laying the groundwork for the subsequent development of your sticky notes application. Through practical experience, you will witness first hand how modular design principles manifest in real-world projects, providing a solid foundation for the upcoming stages of your learning journey.

Before we start, we need to create and activate a virtual environment (**venv**). Recall that a virtual environment is a tool in Python to create isolated Python environments. Each virtual environment has its installation directories and doesn't share libraries with other environments or the global Python installation. This isolation is particularly useful for managing dependencies and avoiding conflicts between different projects. You may refer back to the **Project Setup Guide** if you need to revise the steps required to set up a virtual environment with `venv`.

Once you have created and activated a new virtual environment for this project, we can move on to installing Django within the new virtual environment. Install Django by entering the following command in your terminal using the `pip` package manager:

```
pip install django
```

When working with Django in the future, always make sure to first activate the virtual environment where Django is installed.

Step 1: Create a Django project and application

Imagine you have been tasked with creating a bulletin board application to share information, announcements, and resources across an organisation. A post should have a title, body, and author name fields to be completed by the author. Then, the current date and time should be attached to the post when it is shared.

To begin, you should navigate to the directory where you want to create the Django project. Then, you can start a new Django project called **bulletin_board** using the following command:

```
django-admin startproject bulletin_board
```

After running the above command, you will see a new project directory called **bulletin_board** in the directory where you executed the command. Inside, there should be a folder with the same name. This inner folder is the **core app** of your project, used for configuring your Django application and linking together the rest of the applications. Below is the file structure of the directory:

```
bulletin_board/  (This is your project directory)
    manage.py
  bulletin_board/  (This is your core app)
      __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

Extra resource

Explore the Django documentation to learn more about what is created with the **`startproject` command**.

Next, navigate to the project directory by changing the directory in the command line:

```
cd bulletin_board
```

Ensure you are in the project directory and not in the core app folder that shares the same name as the project directory before running the next command. To create a new Django app called **posts**, enter the following into your command line:

```
python manage.py startapp posts
```

Here, `manage.py` is a command-line utility that comes with Django, providing a set of commands for performing various tasks related to Django projects. The `manage.py` script is automatically created when you start a new Django project using the `django-admin startproject` command.

Step 2: Install your app in settings.py

In **bulletin_board/settings.py**, install your application:

```
# Application definition
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "Django.contrib.staticfiles",

    # Your newly created applications go here
    "posts",
]
```

In **settings.py** you will configure settings such as database connection, static files, middleware, applications, etc. Middleware is a type of software that provides common services and capabilities to applications outside of what's offered by the operating system. It acts as a bridge/pipeline between application software and the underlying operating system, network, or database, and facilitates communication and data management for distributed applications. It can also manage tasks like messaging, authentication, and API management, making it easier for different components of a system to communicate and function together effectively.

Static files include assets such as CSS files, JavaScript files, images, fonts, and other resources that don't change dynamically. These files are typically served directly by the server and don't require any processing by your application's server-side code. Django's settings file (**settings.py**) holds the global configuration.

Step 3: Define model

In **posts/models.py**, define a model for the posts:

```python
# posts/models.py
from django.db import models

class Post(models.Model):
    """Model representing a bulletin board post.

    Fields:
    - title: CharField for the post title with a maximum length of 255
characters.
    - content: TextField for the post content.
    - created_at: DateTimeField set to the current date and time when the
post is created.

    Relationships:
    - author: ForeignKey representing the author of the post.

    Methods:
    - __str__: Returns a string representation of the post, showing the
title.

    :param models.Model: Django's base model class.
    """

    title = models.CharField(max_length=255)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    # Define a ForeignKey for the author's relationship
    author = models.ForeignKey(
        "Author", on_delete=models.CASCADE, null=True, blank=True
    )

    def __str__(self):
        return self.title


class Author(models.Model):
    """
    Model representing the author of a bulletin board post.

    Fields:
    - name: CharField for the author's name.
    Methods:
    - __str__: Returns a string representation of the author, showing the
name.

    :param models.Model: Django's base model class.
    """

    name = models.CharField(max_length=255)

    def __str__(self):
        return self.name
```

Models serve as a means of representing real-world entities and their relationships within a database. In this section, the `Post` model incorporates primary and foreign keys, reflecting the inherent relationships between business entities in the bulletin board application.

`Post` model refinement:

Fields: The `Post` model maintains three fundamental fields:

- `title`: A `CharField` capturing the post title with a maximum length of 255 characters.
- `content`: A `TextField` designed for more extensive post content.
- `created_at`: A `DateTimeField` which automatically records the date and time when a new post instance is created.

Relationships: The model includes a `ForeignKey` relationship, introducing a connection to the `Author` model. The `author` field establishes a link between a post and its author.

`Author` model introduction:

A new model, `Author`, is introduced to represent the authors of the bulletin board posts.

Fields: The `Author` model is equipped with a single field:

- `name`: A `CharField` capturing the author's name.

Relationships: No specific relationships are defined within the `Author` model, as its purpose is to complement the `Post` model's foreign key relationship.

Methods: No specific methods are implemented in either model, keeping the focus on data representation.

By incorporating a `ForeignKey` relationship, the `Post` model now establishes a connection with the `Author` model, enabling each post to be associated with a specific author. The `on_delete=models.CASCADE` parameter ensures that if an author is deleted, all corresponding posts will also be removed, maintaining data integrity.

Step 4: Run migrations

Ensure you are within the project directory, then run the following command in the terminal:

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

- `python manage.py`: This is the standard way to run management commands in Django. It looks for a file named `manage.py` in your project's main directory.
- `makemigrations`: Running `makemigrations` is the first step in the process of applying changes to your database schema. This command looks at the current state of your models and the existing database schema, and it creates migration files that contain the changes needed to bring the database schema in line with your models.
- `migrate`: This is the specific command to run migrations. When you run this command, Django will examine your models, compare them to the current state of the database, and generate SQL statements to make the database schema match the models.

Step 5: Create views

In **posts/views.py**, define views to handle the creation, viewing, updating, and deletion of posts:

```python
# posts/views.py
from django.shortcuts import render, get_object_or_404, redirect
from .models import Post
from .forms import PostForm


def post_list(request):
    """
    View to display a list of all posts.

    :param request: HTTP request object.
    :return: Rendered template with a list of posts.
    """
    posts = Post.objects.all()

    # Creating a context dictionary to pass data
    context = {
        "posts": posts,
        "page_title": "List of Posts",
    }

    return render(request, "posts/post_list.html", context)


def post_detail(request, pk):
    """
    View to display details of a specific post.

    :param request: HTTP request object.
    :param pk: Primary key of the post.
    :return: Rendered template with details of the specified post.
    """
    post = get_object_or_404(Post, pk=pk)
    return render(request, "posts/post_detail.html", {"post": post})


def post_create(request):
    """
    View to create a new post.

    :param request: HTTP request object.
    :return: Rendered template for creating a new post.
    """
    if request.method == "POST":
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save(commit=False)
            post.save()
            return redirect("post_list")
    else:
        form = PostForm()
    return render(request, "posts/post_form.html", {"form": form})


def post_update(request, pk):
    """
    View to update an existing post.

    :param request: HTTP request object.
    :param pk: Primary key of the post to be updated.
    :return: Rendered template for updating the specified post.
    """
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostForm(request.POST, instance=post)
        if form.is_valid():
            post = form.save(commit=False)
            post.save()
            return redirect("post_list")
    else:
        form = PostForm(instance=post)
    return render(request, "posts/post_form.html", {"form": form})


def post_delete(request, pk):
    """
    View to delete an existing post.

    :param request: HTTP request object.
    :param pk: Primary key of the post to be deleted.
    :return: Redirect to the post list after deletion.
    """
    post = get_object_or_404(Post, pk=pk)
    post.delete()
    return redirect("post_list")
```

Views play a crucial role in responding to user requests and implementing the core logic of the application.

`post_list` view:

- Retrieves all posts from the database.
- Constructs a context dictionary containing the posts and additional data if needed.
- Renders the `post_list.html` template, presenting a list of posts.

`post_detail` view:

- Retrieves a specific post from the database based on its primary key.
- Constructs a context dictionary containing the detailed post information.
- Renders the `post_detail.html` template, displaying the individual post details.

`post_create` view:

- Handles the creation of a new post.
- If the request method is `POST`, it validates the form data and saves the new post.
- If the method is `GET`, it displays the form for creating a new post.
- Utilises a context dictionary to pass the form to the template for rendering.

`post_update` view:

- Handles the update of an existing post.
- If the request method is `POST`, it validates the form data and updates the post.
- If the method is `GET`, it displays the form pre-filled with the existing post data.
- Utilises a context dictionary to pass the form and existing post data to the template for rendering.

`post_delete` view:

- Handles the deletion of an existing post.
- Retrieves the specified post based on its primary key.
- Deletes the post from the database.
- Redirects to the post list view after deletion.

These views collectively define the behaviour of the bulletin board application, ensuring that users can interact with the system by listing, viewing, creating, updating, and deleting posts. Additionally, the use of context dictionaries enhances the flexibility of these views by facilitating the transmission of data from the views to the corresponding templates.

Step 6: Create forms

Create a forms file **posts/forms.py** to handle form input:

```python
# posts/forms.py
from django import forms
from .models import Post


class PostForm(forms.ModelForm):
    """
    Form for creating and updating Post objects.

    Fields:
    - title: CharField for the post title.
    - content: TextField for the post content.

    Meta class:
    - Defines the model to use (Post) and the fields to include in the
form.

    :param forms.ModelForm: Django's ModelForm class.
    """

    class Meta:
        model = Post
        fields = ["title", "content", "author"]
```

Forms in Django handle user input. In this section, a form named `PostForm` is created based on the `Post` model, and the `forms.ModelForm` is a helper class that defines the form's behaviour based on a model. The form includes fields for the posts' `title`, `content`, and `author`.

Step 7.1: Configure URL patterns

Create a **posts/urls.py** file to handle the URL patterns for your project.

```python
# posts/urls.py
from django.urls import path
from .views import (
    post_list,
    post_detail,
    post_create,
    post_update,
    post_delete,
)

urlpatterns = [
    # URL pattern for displaying a list of all posts
    path("", post_list, name="post_list"),

    # URL pattern for displaying details of a specific post
    path("post/<int:pk>/", post_detail, name="post_detail"),

    # URL pattern for creating a new post
    path("post/new/", post_create, name="post_create"),

    # URL pattern for updating an existing post
    path("post/<int:pk>/edit/", post_update, name="post_update"),

    # URL pattern for deleting an existing post
    path("post/<int:pk>/delete/", post_delete, name="post_delete"),
]
```

URL patterns define the mapping between URLs and views. Each URL pattern is associated with a specific view function or class, which handles the request and returns a response. The `<int:pk>` is a path converter that matches an integer and passes it as the `pk` parameter to the associated view. Name attributes (`name="post_list"`) are used to provide names for the URL patterns, making it easier to reference them in the code, such as in templates.

Step 7.2: Include URLs in your main urls.py

```python
# bulletin_board/urls.py
from django.contrib import admin
from django.urls import path, include

# Define URL patterns for the entire project
urlpatterns = [
    # Admin URL pattern, mapping to the Django admin interface
    path("admin/", admin.site.urls),

    # Include URL patterns from the 'posts' app
    # All URLs from 'posts.urls' will be prefixed with '/'
    path("", include("posts.urls")),
]
```

By adding the `include` statement `path("", include("posts.urls"))`, you are telling Django to include all the URL patterns defined in **posts/urls.py** under the root URL. This setup ensures that URLs defined in your app are recognised by Django and routed to the appropriate views.

Step 8: Create a templates folder

Before continuing with the next steps, you will need to create a **templates** folder within the posts app. Templates in Django are HTML files that are used to define the structure and layout of your web pages. The templates combine HTML with Django's template language to include variables, control structures, and reusable components to dynamically render content.

Set this up within the posts app by creating a folder named **templates**, if it doesn't already exist. Note that the folder has to be named "templates". This directory will store all the HTML template files related to the posts app.

Step 8.1: Base template

After creating the **templates** folder within the posts app, we can create our first template. Begin by creating a file named **base.html** inside the **templates** folder. The **base.html** template will define the common structure of the site, including essential elements such as the header, footer, and navigation bar. Creating a **base.html** file can allow us to create a more consistent layout across each of our pages by extending the other templates from the **base.html** template.

```html
<!-- posts/templates/base.html -->
{% load static %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>
      {% block title %}
        Bulletin Board
      {% endblock title %}
    </title>
  </head>
  <body>
    <header>
      <h1>
        {% block header %}
          Bulletin Board
        {% endblock header %}
      </h1>
      <!-- Add navigation links or other header content here -->
    </header>
    <main>
      {% block content %}
      {% endblock content %}
    </main>
    <footer>
      <!-- Add footer content or links here -->
    </footer>
  </body>
</html>
```

Because we are using `{% load static %}` files in our template we will need to update our **bulletin_board/settings.py** file:

```python
# bulletin_board/settings.py

# Static files (CSS, JavaScript, Images)
STATIC_URL = 'static/'

# Add static files directories
STATICFILES_DIRS = [
    BASE_DIR / "posts/static",
    # Add other app static directories if needed
]

# Define the root directory for static files
STATIC_ROOT = BASE_DIR / "static"
```

Let's break down the specific lines related to static files:

- `STATIC_URL = 'static/'`: This setting defines the base URL for serving static files. In this case, any static files (like CSS, JavaScript, or images) will be served under the URL path `static/`.
- `STATICFILES_DIRS`: This setting is a list of directories where Django will look for additional static files. In this case, it includes the directory `BASE_DIR / "posts/static"`. This is useful when you have static files associated with your Django apps and you want Django to collect them during the `collectstatic` management command.
- `STATIC_ROOT = BASE_DIR / "static"`: This setting specifies the directory where Django will collect and store all the static files from your various apps and `STATICFILES_DIRS`. During development, Django serves static files directly from their respective locations, but in a production environment, you typically collect all static files into a single directory for efficient serving. The **`collectstatic`** management command is used to gather these files.

Now, when you create other templates, you can extend the **base.html** template and fill in the content for specific blocks.

Step 8.2: Create templates

For the following HTML templates, create each template within the **posts/templates/posts/** folder. To create the subfolder, inside the **templates** folder create a new directory named **posts**. This subfolder will be used to store all the HTML template files specific to the posts app.

After creating the next few templates, the posts app should have a **templates** folder structure similar to the below example:

```
posts/
├── templates/
│   ├── posts/
│   │   ├── post_list.html
│   │   ├── post_detail.html
│   │   └── post_form.html
```

Template for listing posts (post_list.html):

```
<!-- posts/templates/posts/post_list.html -->
{% extends 'base.html' %}
{% block title %}
  Bulletin Board - {{ page_title }}
{% endblock %}
{% block content %}
  <h2>
    {{ page_title }}
  </h2>
  <ul>
    <a href="{% url 'post_create' %}">Add post</a>
    {% for post in posts %}
      <li>
        <a href="{% url 'post_detail' pk=post.pk %}">{{ post.title
}}</a>
        <p>
          {{ post.content }}
        </p>
      </li>
    {% endfor %}
  </ul>
{% endblock %}
```

Template for viewing post details (post_detail.html):

```
<!-- posts/templates/posts/post_detail.html -->
{% extends 'base.html' %}
{% block title %}
  Bulletin Board - {{ post.title }}
{% endblock %}
{% block content %}
  <h2>
    {{ post.title }}
  </h2>
  <p>
    {{ post.content }}
  </p>
  <p>
    Created at: {{ post.created_at }}
  </p>
  <a href="{% url 'post_update' pk=post.pk %}">Update Post</a>
  <a href="{% url 'post_delete' pk=post.pk %}">Delete Post</a>
  <a href="{% url 'post_list' %}">Back to Post List</a>
{% endblock %}
```

Template for creating and updating posts (post_form.html):

```
<!-- posts/templates/posts/post_form.html -->
{% extends 'base.html' %}
{% block title %}
  Bulletin Board -
  {% if form.instance.pk %}
    Edit
  {% else %}
    Create
  {% endif %}
  Post
{% endblock %}
{% block content %}
  <h2>
    {% if form.instance.pk %}
      Edit
    {% else %}
      Create
    {% endif %}
    Post
  </h2>
  <form method="post"
        action="{% if form.instance.pk %}{% url 'post_update'
pk=form.instance.pk %}{% else %}{% url 'post_create' %}{% endif %}">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">
      Save
    </button>
  </form>
  <a href="{% url 'post_list' %}">Back to Post List</a>
{% endblock %}
```

Template tags in Django are special syntax enclosed like this **`{% ... %}`** or like this **`{{ ... }}`**. They are used within Django templates to embed dynamic content or control the template's logic, and they allow you to execute Python-like code directly within your HTML templates, for example: `<p>Hello, {{ user.username }}!</p>`.

These templates extend the **base.html** template to maintain a consistent layout.

- `{% block title %}` is used to dynamically set the page title.
- `{% block content %}` is where the specific content for each page is placed.
- For the form, `{{ form.as_p }}` renders the form fields in a paragraph format.

Step 8.3: Creating CSS

In your Django app posts, create a folder named **posts/static/posts**, then create a CSS file called **posts/static/posts/styles.css**:

```
/* posts/static/posts/styles.css */

body {
  font-family: "Arial", sans-serif;
}

header {
  background-color: #333;
  color: white;
  padding: 10px;
  text-align: center;
}
```

```
main {
  margin: 20px;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  margin-bottom: 20px;
}

footer {
  background-color: #333;
  color: white;
  padding: 10px;
  text-align: center;
}
```

Update your HTML templates to link to the CSS file. For example, in your **base.html**:

```
<!-- posts/templates/base.html -->
{% load static %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>
      {% block title %}
        Bulletin Board
      {% endblock title %}
    </title>
    <!-- Add your CSS links or stylesheets here -->
    <link rel="stylesheet" href="{% static 'posts/styles.css' %}" />
  </head>
  <body>
    <header>
      <h1>
```

```
        {% block header %}
          Bulletin Board
        {% endblock header %}
      </h1>
      <!-- Add navigation links or other header content here -->
    </header>
    <main>
      {% block content %}
      {% endblock content %}
    </main>
    <footer>
      <!-- Add footer content or links here -->
    </footer>
  </body>
</html>
```

Example project structure after creating the templates:

```
bulletin_board/
├── manage.py
├── posts/
│   ├── migrations/
│   ├── static/
│   │   ├── posts/
│   │   │   └── styles.css
│   ├── templates/
│   │   ├── posts/
│   │   │   ├── post_list.html
│   │   │   ├── post_detail.html
│   │   │   └── post_form.html
│   │   └── base.html
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── models.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── bulletin_board/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── db.sqlite3
```

Step 8.4: Collect static files

Django needs to collect static files to serve them during development. Run the following in the command line:

```
python manage.py collectstatic
```

Step 9: Run migrations and start the server

Then, run the following in the command line:

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

- The `makemigrations` command in Django is used to create new database migration files based on changes made to your Django models. When you create or modify a model, running `makemigrations` analyses the changes and produces migration files in the **migrations** directory of your app.
- The `migrate` command in Django is responsible for applying or undoing database migrations. When executed, Django reads the migration files in the **migrations** directory and updates the database schema to reflect the changes made to your models. Running `migrate` is a crucial step after creating or modifying models using `makemigrations` to ensure that the database structure aligns with the current state of your Django project.
- The development server is started using the `runserver` command, allowing you to test your application locally.

You can visit the specified URLs (e.g., **http://127.0.0.1:8000/**) to interact with the application. At this stage, you will not see any posts listed, but we will address this soon by adding posts. You can quit the server with CTRL+C or CMD+C in the command line.

Step 10: Creating posts for the bulletin board app

Register your models in the **posts/admin.py** file:

```python
# posts/admin.py

from django.contrib import admin
from .models import Post
from .models import Author

# Register your models here.

# Post model
admin.site.register(Post)

# Author model
admin.site.register(Author)
```

This script is used to register the `Post` and `Author` models with the Django admin interface. It starts by importing necessary modules and models: `admin` from `django.contrib`, and `Post` and `Author` from the current app's models.

`Post` model registration: `admin.site.register(Post)` registers the `Post` model, making it accessible in the Django admin panel. It allows administrators to manage (add, edit, and delete) posts.

`Author` model registration: `admin.site.register(Author)` registers the `Author` model. This adds the model to the admin panel, enabling the management of author data.

Create a superuser:

If you want to access the Django admin interface, you can create a superuser account with the following command in your terminal:

```
python manage.py createsuperuser
```

1. In username add `admin`.
2. In email, add your email address, such as `example@email.com`.
3. In password, type your password.
4. In password (again) retype your password.

Note: When creating a superuser the password field will remain blank while you are typing the password, but Django is in fact accepting your input.

Step 11: Run migrations and start the server

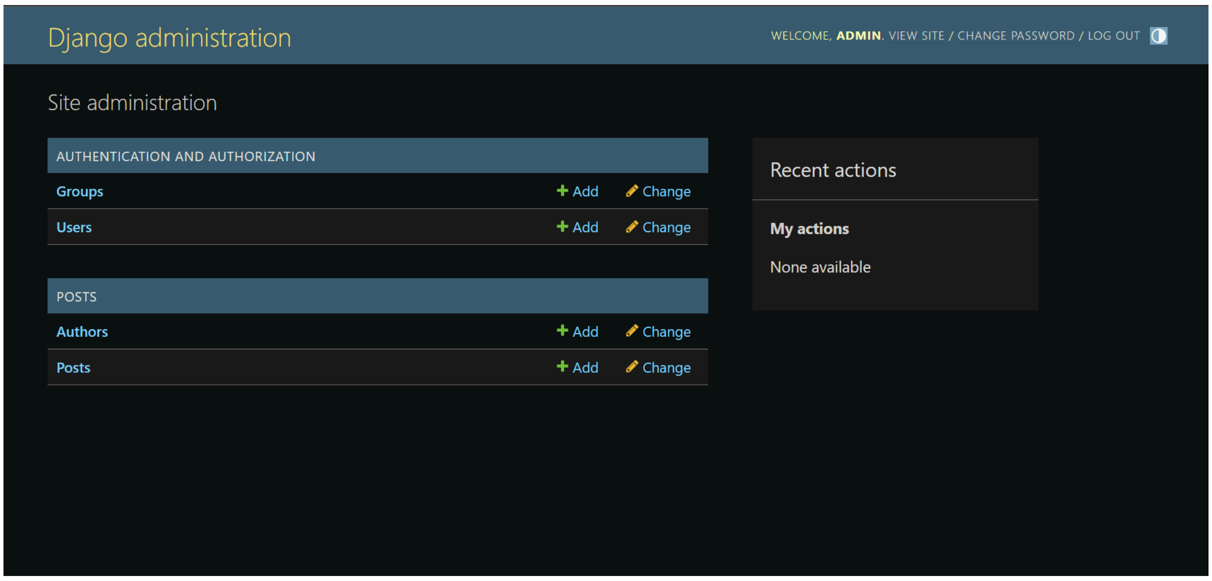
Run the following in your command line in the project directory:

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

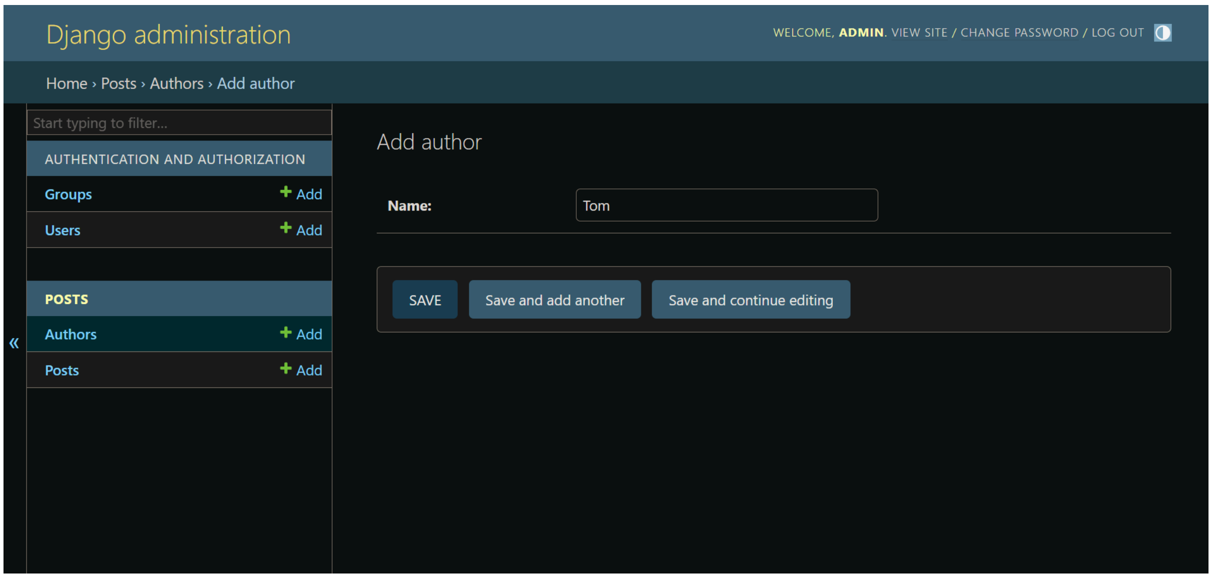
By default, it will run on **http://localhost:8000/**.

Access the admin interface:

Once you have created a superuser, you can access the admin interface at **http://localhost:8000/admin/** and log in with the superuser credentials. Once logged in, you will be presented with this screen:

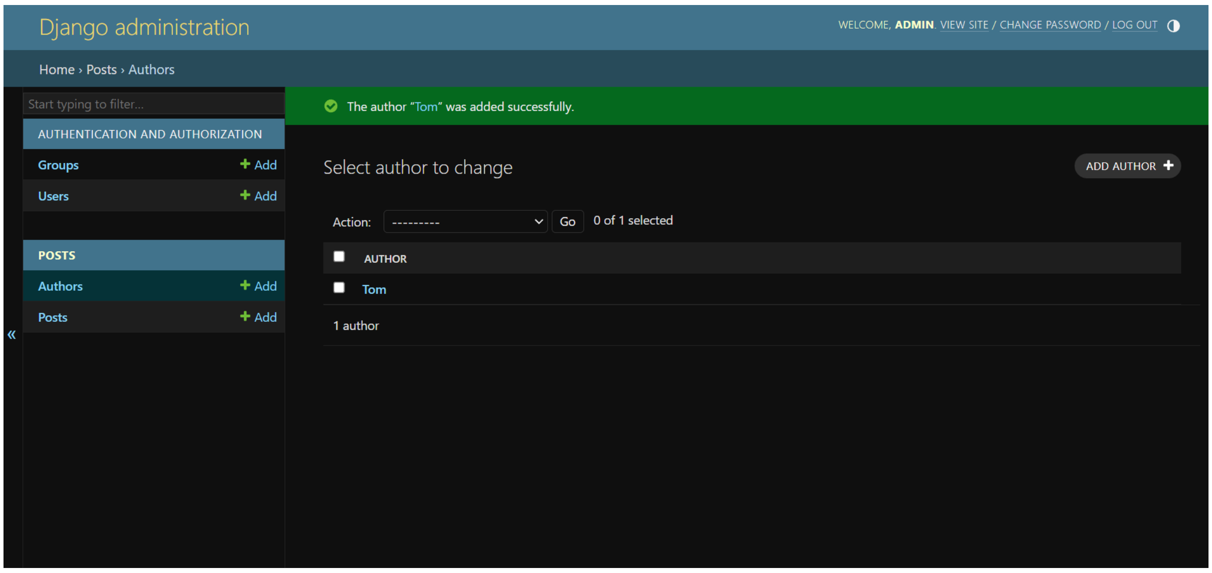


Now we can start by creating an author for our first post by pressing on the "**Add**" button for the author row. Doing so will allow you to add author details similar to the example below:


Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Posts › Authors › Add author
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POSTS
Authors + Add
Posts + Add
Add author
SAVE
Save and add another
Save and continue editing

Name: Tom

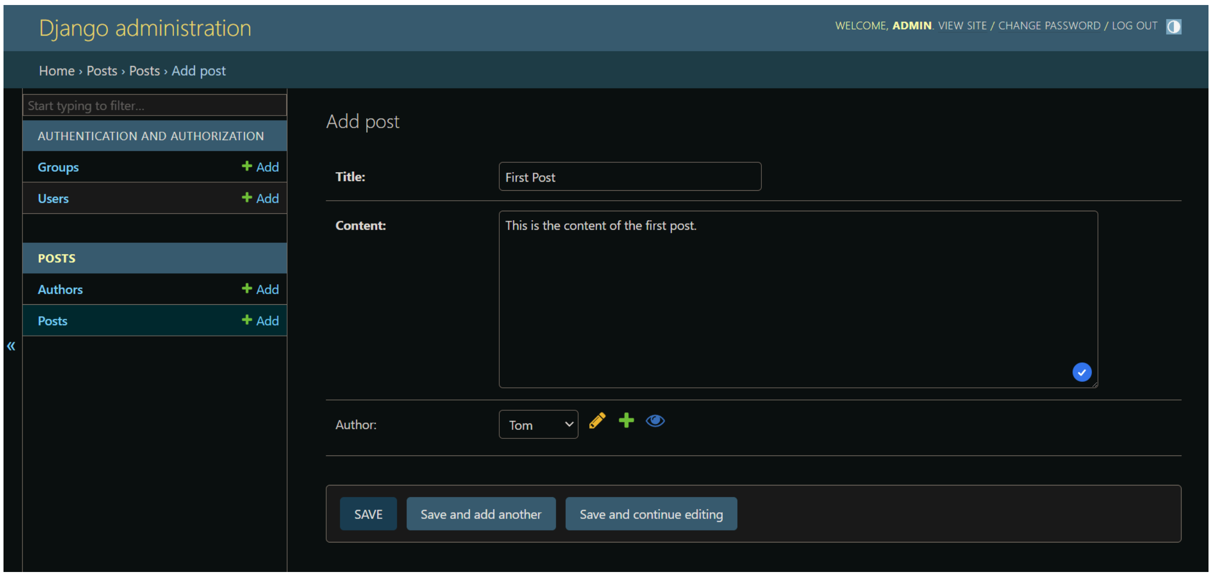
After using the save button to add our first author, you should be presented with a success page similar to the example below:


Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Posts › Authors
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POSTS
Authors + Add
Posts + Add
ADD AUTHOR
Action: --------- Go 0 of 1 selected
AUTHOR
Tom
1 author

The author "Tom" was added successfully.

Select author to change

After adding our first author, we can continue to create our first post:


Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Posts › Posts › Add post
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POSTS
Authors + Add
Posts + Add
SAVE
Save and add another
Save and continue editing

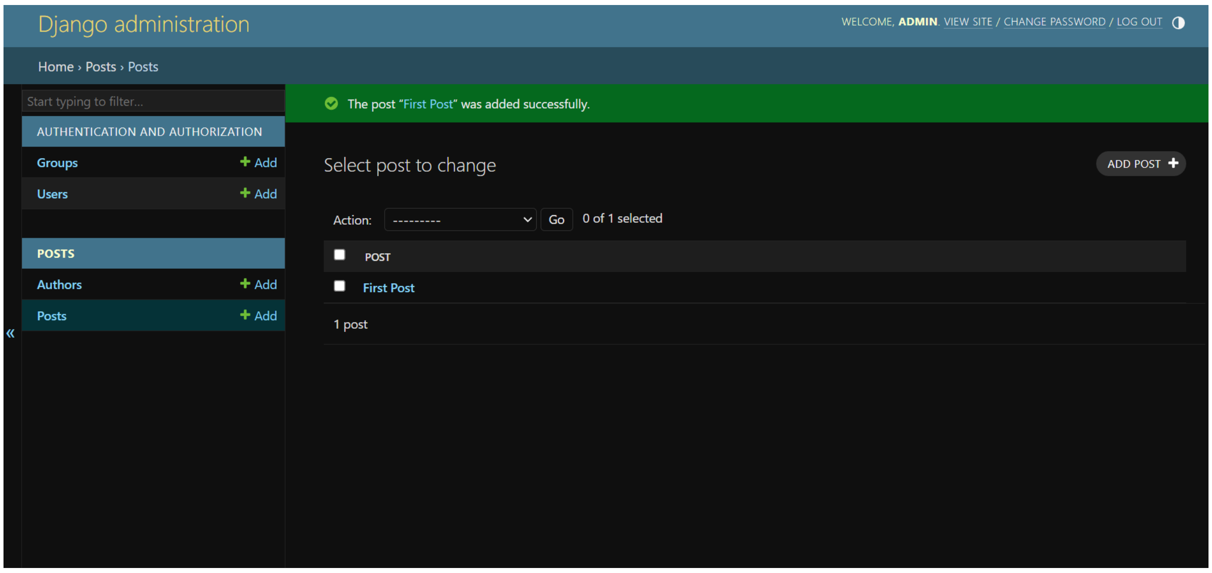
Add post

Title: First Post

Content: This is the content of the first post.

Author: Tom

After using the save button to add our first post, you should be presented with a success page similar to the example below:



The post "First Post" was added successfully.

Select post to change

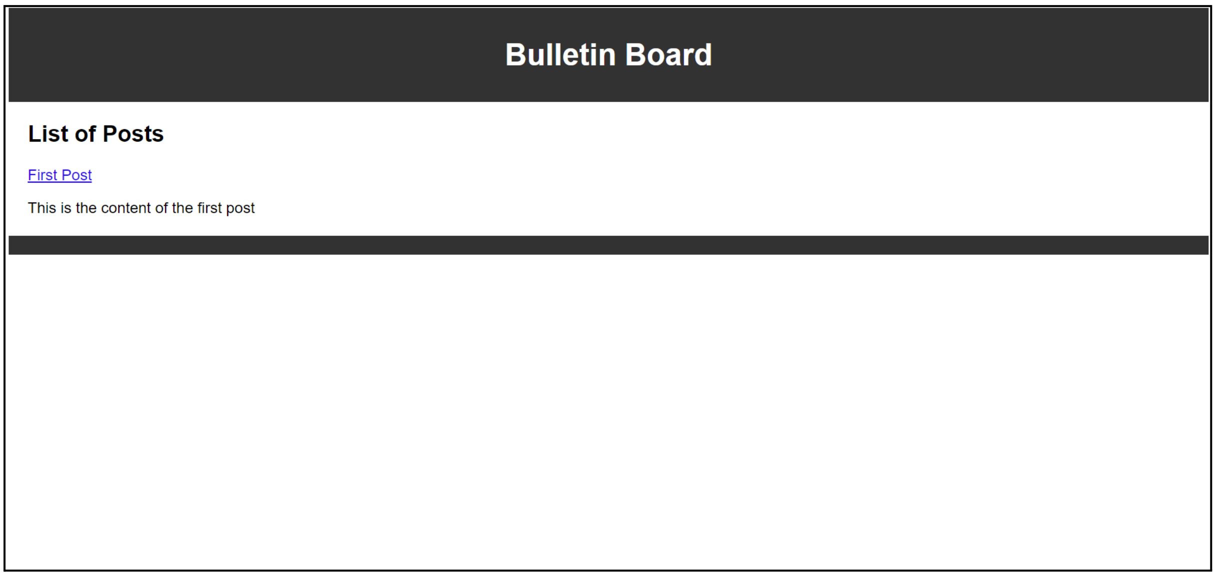
0 of 1 selected

POST
First Post

1 post

Log out at the top right of your screen and navigate back to **http://localhost:8000/**.

Your new post will be visible, as shown below:



Bulletin Board

List of Posts

First Post

This is the content of the first post

Instructions

This practical task is designed to test your knowledge of Django by building a sticky notes application. You will be focusing on the creation of a Django project, defining the model, running migrations, and implementing views and forms. You will also focus on the configuration of URL patterns, creating templates, styling with CSS, collecting static files, running migrations, and creating posts for the sticky notes application.

Remember that in creating your sticky notes application with Django, you will need to demonstrate your understanding of the Model-View-Template (MVT) architecture and implement CRUD functionality. Also, ensure your code adheres to PEP 8 styling guidelines. Let's get started!

Practical task

Part 1

Follow these steps:

1. Use the design techniques and principles you have learnt in the bootcamp to design your sticky notes application. Please remember to submit your diagrams alongside your project. Feel free to review the Software Design task if you need a refresher. You will need to submit at least the following:
    - Use case diagram
    - Sequence diagram
    - Class diagram
2. Create a new Django project called **sticky_notes**.
3. Install your app in **settings.py**. Configure your Django project settings to include the newly created app.
4. Design and define the model for your sticky notes application, including fields for the title and content of each note.
5. Implement and run database migrations to ensure that your model is reflected in the database.
6. Develop views that handle the CRUD functionality for your sticky notes application, allowing users to create, read, update, and delete notes.
7. Build forms that facilitate the input and editing of sticky notes content.
8. Set up URL patterns to map the different views of your sticky notes application. Ensure that the URLs of your sticky notes app are appropriately included in the main **urls.py** file.
9. Create a base HTML template that can be extended by other templates in your application.
10. Design HTML templates for displaying sticky notes content, integrating them with your views.
11. Style your sticky notes application by adding CSS to enhance the visual appeal. If you are familiar with Bootstrap you are welcome to use it to speed up your CSS development.

12. Configure your Django project to collect static files, including CSS, into a designated directory.
13. Rerun migrations to ensure that any changes are reflected in the database, and start the development server to test your application.
14. Please **exclude** the `venv` folder when you submit your Django project; it is unnecessary, as the mentor reviewing your work can generate it. It is good practice to always delete generated files since they may make it difficult to set up your app on other machines and they take up a lot of storage space.

Part 2

Create a text document, markdown file, or PDF called **research_answers** to record your findings for the following:

1. Conduct research to understand how HTTP applications preserve the state of an application across multiple request-response cycles, especially concerning user authentication and session management. Write a paragraph (150–250 words) describing your findings.
2. Investigate and document the procedures for performing Django database migrations to a server-based relational database like MariaDB. Write a paragraph (150–250 words) describing your findings.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.