MODULE-2: DJANGO TEMPLATES AND MODELS

Django templates and models are two fundamental components of Django web applications, each serving a distinct purpose in the development process. Let's explore each of them:

Django Templates:

- Django templates are used to generate dynamic HTML content for web pages.
- They are text files containing HTML markup with embedded Django template language code.
- Template files typically have the extension .html.
- Django's template engine processes these files, replacing template variables, tags, and filters with actual values.
- Templates allow for the separation of presentation logic from business logic, promoting cleaner and more maintainable code.
- Template inheritance enables the creation of reusable layouts and the extension of base templates to create specialized pages.
- Template tags and filters provide additional functionality for conditionals, loops, formatting, and more.
- Templates are rendered by views and returned as HTTP responses to client requests.

Django Models:

- Django models represent the structure and behavior of data in a Django application.
- They are Python classes that inherit from **django.db.models.Model**.
- Each model class corresponds to a database table, with attributes representing fields in the table.
- Django's **ORM** (Object-Relational Mapping) translates model classes and their relationships into database schema and queries.
- Models define relationships such as ForeignKey, OneToOneField, and ManyToManyField to establish connections between data entities.
- Model fields specify the data type and constraints for each attribute, such as CharField, IntegerField, DateField, etc.
- Django's migration system (manage.py makemigrations and manage.py migrate) manages changes to the database schema based on model definitions.
- Models encapsulate data access and manipulation logic, providing an object-oriented interface to interact with the underlying database.
- They support validation, querying, and CRUD (Create, Read, Update, Delete) operations on data entities.

In summary, Django templates facilitate the generation of dynamic HTML content for web pages, while Django models define the structure and behavior of data entities in a Django application. Together, they form the backbone of Django web development, enabling the creation of robust and scalable web applications.

2.1 TEMPLATE SYSTEM BASICS

Django's template system is designed to separate the presentation layer (HTML templates) from the business logic (Python code) in web applications. This separation allows developers to focus on designing the user interface without mixing it with the application's backend logic.

Syntax :

Django's template syntax consists of special tags, variables, and filters enclosed within

`{% ... %}`, `{{ ... }}`, and `{{ ... | filter }}` respectively.

You can render dynamic content in templates using variables passed via a context dictionary. Let's consider an example:

pip install virtualenvwrapper-win

mkvirtualenv mytest

pip install Django

django-admin --version

django-admin startproject helloproj

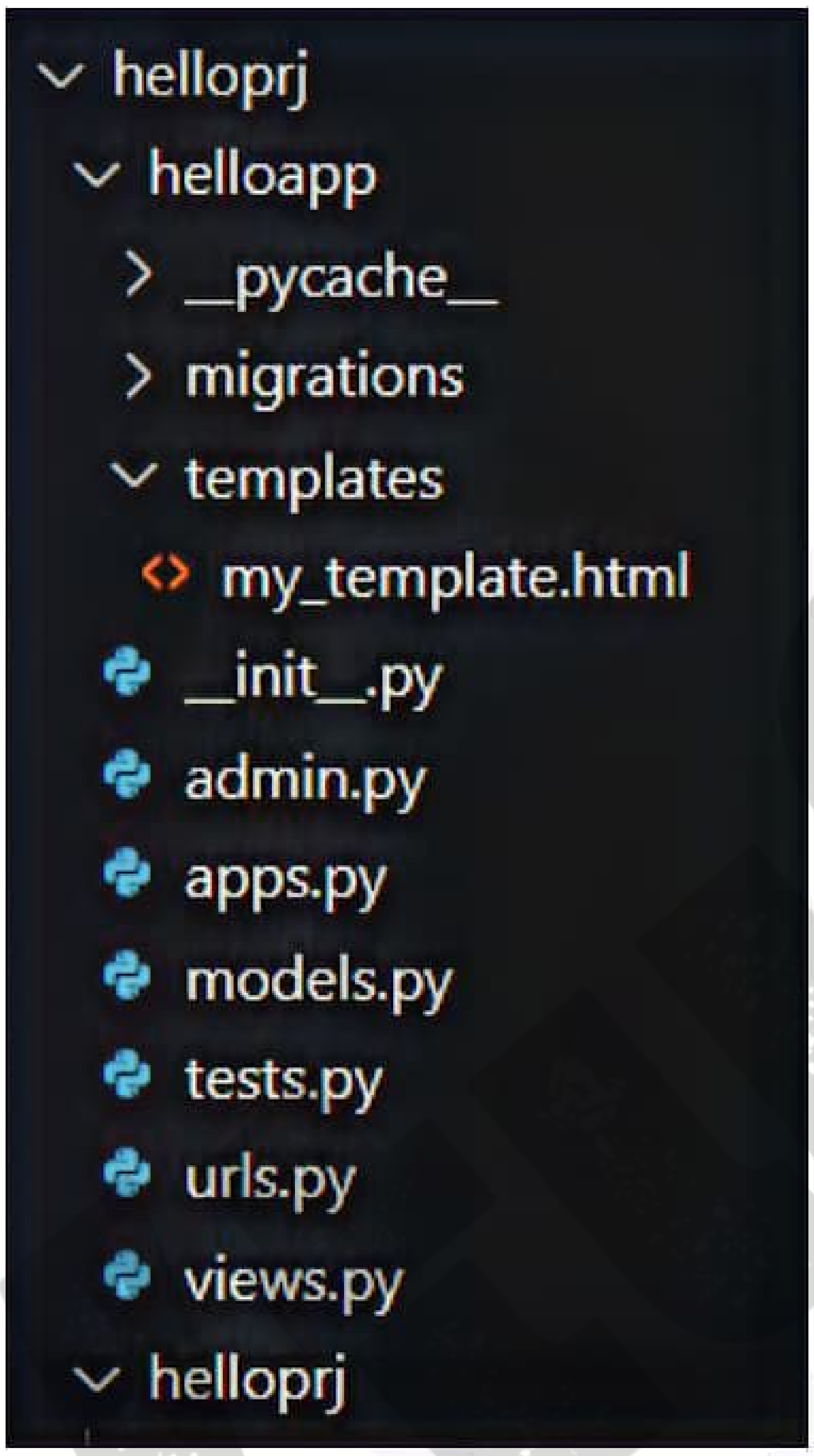
django-admin startapp helloapp

modify the setting.py in helloproj

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

urls.py

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

from django.urls import include


urlpatterns = [

   path('', include('helloapp.urls')),

   path('admin/', admin.site.urls),

 ]
```



```
helloprj
  helloapp
    > __pycache__
    > migrations
    templates
      my_template.html
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    urls.py
    views.py
  helloprj
```

You can render dynamic content in templates using variables passed via a context dictionary.

Views.py

```
from django.shortcuts import render

from django.http import HttpResponse

# Create your views here.

def my_view(request):

   context = {
    'username': 'Sachin Tendulkar',

    'age': 45,

    'is_registered': True

  }

  return render(request, 'my_template.html', context)
```

In the above example, the `context` dictionary contains data like `username`, `age`, and `is_registered`, which can be accessed in the template.

create folder templated/my_template.html

```
<!-- my_template.html -->

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <title>User Profile</title>

</head>

<body>

  <h1>Hello, {{ username }}!</h1>

  <p>You are {{ age }} years old.</p>

  {% if is_registered %}

    <p>Welcome back!</p>

  {% else %}

    <p>Please register to continue.</p>

  {% endif %}

</body>

</html>
```

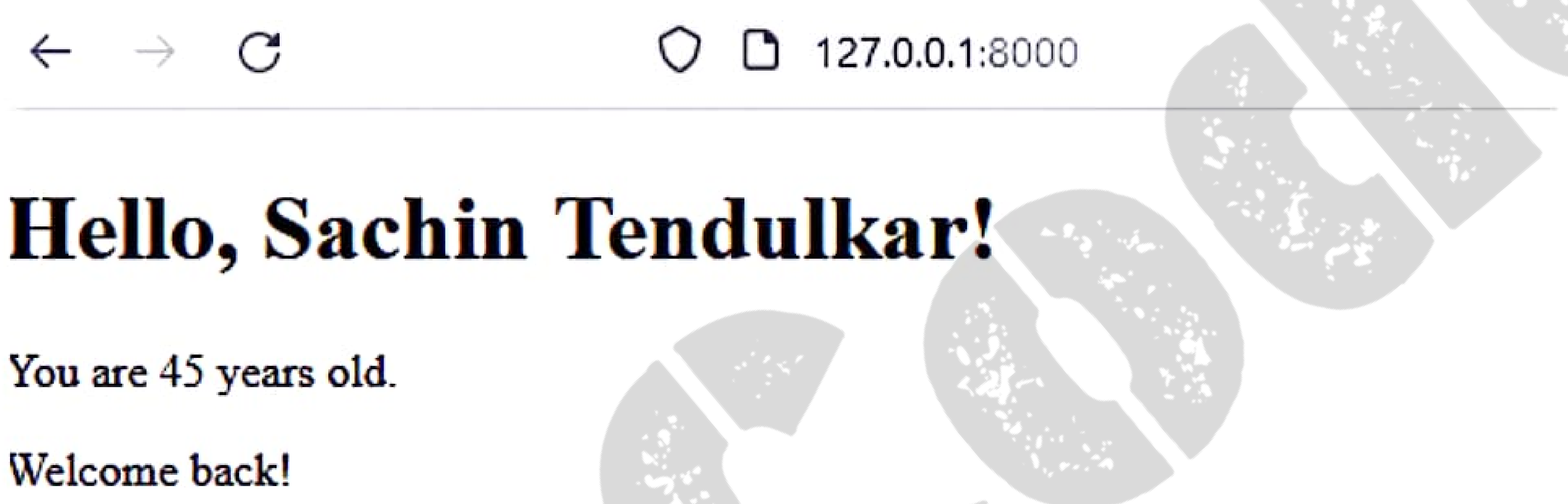
The `context` dictionary passed to the `render` function contains data that will be available in the template. Each key-value pair in the context becomes a variable accessible in the template.

py manage.py migrate

Running `python manage.py migrate` is an essential step when you make changes to your project's models or when you initially set up your project. It ensures that your database schema stays in sync with your project's models, preventing inconsistencies and errors when interacting with the database.

python manage.py runserver

Output:



127.0.0.1:8000

Hello, Sachin Tendulkar!

You are 45 years old.

Welcome back!

2.2 USING THE DJANGO TEMPLATE SYSTEM

Django template system involves creating HTML templates that Django can dynamically render with data passed from views. Here's a guide to using the Django template system:

1. Create Templates:

Create HTML templates in your Django project's template directory. By default, Django looks for templates in a directory named templates within each app and in the project's templates directory.

```
<!-- example_template.html -->

<!DOCTYPE html>

<html lang="en">

<head>

   <meta charset="UTF-8">

   <title>{{ page_title }}</title>

</head>
```

```
<body>
  <h1>{{ heading }}</h1>
  <p>{{ content }}</p>
</body>
</html>
```

2. Define Views:

Define views in your Django app's `views.py` file. Views are Python functions that handle HTTP requests and return HTTP responses, typically by rendering a template.

```
# views.py
from django.shortcuts import render


def my_view(request):
  context = {
    'page_title': 'Welcome to My Website',
    'heading': 'Hello, Django!',
    'content': 'This is a sample Django template.',
  }
  return render(request, 'example_template.html', context)
```

3. Render Templates in Views:

Use the `render` function from `django.shortcuts` to render templates in views. Pass the request, template name, and context data as arguments to the `render` function.

4. Handle URLs:

Map URLs to views in your project's URL configuration (`urls.py`). Define URL patterns using the `path` function and specify the corresponding view function.

```
# urls.py
from django.urls import path
from .views import my_view
urlpatterns = [
    path('my-view/', my_view, name='my_view'),
]
```

5. Accessing Data in Templates:

Access data passed from views in templates using template variables enclosed within double curly braces (`{{ ... }}`). These variables are replaced with actual data when the template is rendered.

6. Control Flow and Logic:

Use template tags like `{% if %}`, `{% for %}`, and `{% block %}` to add control flow and logic to your templates. These tags enable you to perform conditional rendering, iterate over lists, and define template blocks for inheritance.

7. Filters:

Apply filters to template variables using the pipe symbol (`|`). Filters modify the output of variables, allowing you to format data, manipulate strings, and perform other transformations.

8. Template Inheritance:

Implement template inheritance to create reusable templates and organize your template hierarchy effectively. Define a base template with common elements and extend it in child templates to override specific blocks or add additional content.

By following these steps, you can effectively use the Django template system to create dynamic and interactive web pages in your Django projects.

2.3 BASIC TEMPLATE TAGS AND FILTERS

Basic template tags and filters in Django's template system allow you to add logic, control flow, and data manipulation directly within your HTML templates. Let's explore some commonly used template tags and filters:

2.3.1 Template Tags:

1. **{% if %}**: Used for conditional rendering. It allows you to display content based on certain conditions.

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}!</p>
{% else %}
    <p>Please log in to continue.</p>
{% endif %}
```

2. **{% for %}**: Used for looping over lists or querysets to display repetitive content.

```
 <ul>
```

```
{% for item in items %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

3. **{% block %}**: Used in template inheritance to define blocks that child templates can override.

```
<!-- base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}My Website{% endblock %}</title>
</head>
<body>
    {% block content %}
    {% endblock %}
</body>
</html>
```

2.3.2 Template Filters

1. **{{ variable|filter }}**: Filters allow you to modify the output of template variables. Django provides several built-in filters for common tasks.

```
<!-- Convert variable to uppercase -->
{{ name|upper }}

<!-- Truncate text to a certain length -->
{{ description|truncatewords:20 }}

<!-- Format a date -->
{{ date_created|date:"F j, Y" }}
```

2. **safe**: Marks a string as safe HTML, preventing Django from escaping it. Use with caution to avoid XSS vulnerabilities.

```
{{ unsafe_html|safe }}
```

3. **length**: Returns the length of a list, string, or queryset.

```
  {{ items|length }}
```

4. **first**: Returns the first item of a list or queryset.

```
{{ items.first }}
```

These are just a few examples of basic template tags and filters in Django. They provide powerful tools for controlling the presentation of your data and adding dynamic behavior to your HTML templates. You can also create custom template tags and filters to extend Django's template system further.

2.4 MVT DEVELOPMENT PATTERN

The MVT (Model-View-Template) development pattern is Django's interpretation of the popular MVC (Model-View-Controller) architectural pattern. It's a way to structure code in web applications, dividing it into three interconnected components: **Models**, **Views**, and **Templates**.

1. Model:

Models represent the data structure of your application. They encapsulate the data access logic, including querying, insertion, updating, and deletion operations.

Example: Let's consider a simple model representing a blog post. This model would define attributes like title, content, author, publication date, etc.

```
# models.py

from django.db import models

class Post(models.Model):

    title = models.CharField(max_length=100)

    content = models.TextField()

    author = models.ForeignKey(User, on_delete=models.CASCADE)

    publication_date = models.DateTimeField(auto_now_add=True)
```

2. View:

Views handle the presentation logic of your application. They receive HTTP requests from clients, fetch data from the database using models, perform any necessary processing, and then return HTTP responses, typically rendered using templates.

Example: Let's create a view to display a list of blog posts.

```
# views.py

from django.shortcuts import render

from .models import Post
```

```
def post_list(request):

    posts = Post.objects.all()

    return render(request, 'blog/post_list.html', {'posts': posts})
```

3. Template:

Templates handle the user interface of your application. They are HTML files with embedded template tags and variables that Django replaces with actual values when rendering. Templates allow you to generate dynamic content and present it to users.

Example: Create a template to display a list of blog posts.

```
<!-- post_list.html -->

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Blog</title>

</head>

<body>

    <h1>Blog Posts</h1>

    <ul>

    {% for post in posts %}

        <li>{{ post.title }} - {{ post.publication_date }}</li>

    {% endfor %}

    </ul>

</body>

</html>
```

Model: Defines the structure and behavior of the data in your application. It interacts with the database to store and retrieve data.

View: Handles the request-response cycle. It receives requests from clients, interacts with models to fetch data, and then passes that data to templates for rendering.

Template: Defines the presentation layer of your application. It contains HTML markup with embedded template tags and variables, which are replaced with dynamic data when the template is rendered.

In summary, the MVT development pattern in Django promotes a clean separation of concerns, making it easier to manage and maintain complex web applications. Models handle data, views handle logic, and templates handle presentation, allowing for modular, reusable, and maintainable code

let's create a simple Django example using the MVT (Model-View-Template) pattern. We'll build a basic blog application with functionality to display a list of blog posts.

1. Create a Django Project and App:

First, create a new Django project and an app within it.

```
django-admin startproject myblogproject

cd myblogproject

python manage.py startapp blog
```

2. Define Models:

Define a model for the blog post in the blog/models.py file.

```
# blog/models.py

from django.db import models

class Post(models.Model):

    title = models.CharField(max_length=100)

    content = models.TextField()

    author = models.CharField(max_length=50)

    publication_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):

        return self.title
```

3. Create Views:

Create views to handle the request-response cycle in the blog/views.py file.

```
# blog/views.py
```

```
from django.shortcuts import render
from .models import Post
def post_list(request):
    posts = Post.objects.all()
    return render(request, 'blog/post_list.html', {'posts': posts})
```

4. Create Templates:

Create templates to define the presentation layer in the blog/templates/blog directory.

```
<!-- blog/templates/blog/post_list.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Blog</title>
</head>
<body>
    <h1>Blog Posts</h1>
    <ul>
    {% for post in posts %}
        <li>{{ post.title }} - {{ post.author }} ({{ post.publication_date|date:"F j, Y" }})</li>
    {% endfor %}
    </ul>
</body>
</html>
```

5. Define URLs:

Map URLs to views in the blog/urls.py file.

```
# blog/urls.py
from django.urls import path
from .views import post_list

urlpatterns = [

    path('', post_list, name='post_list'),

]
```

6. Include App URLs:

Include the app's URLs in the project's main URL configuration in the myblogproject/urls.py file.

```
# myblogproject/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('blog.urls')),

]
```

7. Run Migrations:

Apply database migrations to create the necessary database tables for the models.

```
python manage.py makemigrations

python manage.py migrate
```

8. Run the Development Server:

Run the Django development server to see the application in action.

```
python manage.py runserver
```

Visit http://127.0.0.1:8000/ in your web browser to see the list of blog posts.

This example demonstrates the basic usage of the MVT pattern in Django to create a simple blog application. Models define the data structure, views handle the logic, and templates handle the presentation of the application.

2.5 TEMPLATE LOADING

Template loading in Django refers to how Django finds and loads templates to render when processing a request. By default, Django looks for templates in specific directories within each app and in the project's global template directories. Let's explore how template loading works:

1. Template Directories:

Django searches for templates in the following directories by default:

Within each app:

Create a directory named templates within each app directory to store app-specific templates.

Project-wide templates:

Optionally, you can define a global templates directory at the project level.

2. Template Loading:

Django uses a template loader to find and load templates from these directories. The template loader searches for templates in the specified directories based on the order defined in the TEMPLATES setting in your project's settings.py file.

3. TEMPLATES Setting:

The TEMPLATES setting in your settings.py file configures how Django handles templates. It includes a list of template engines and their configurations, as well as options for template directories and other settings.

Example:

Here's an example of how you might configure the TEMPLATES setting in your settings.py file:

```
# settings.py

TEMPLATES = [

  {

    'BACKEND': 'django.template.backends.django.DjangoTemplates',

    'DIRS': [

      # List of template directories

      os.path.join(BASE_DIR, 'templates'),  # Project-wide templates directory

      # Additional directories for specific apps, if needed

    ],

    'APP_DIRS': True,

    'OPTIONS': {
        # Other template engine options
    },
  },
]
```

In this example:

'DIRS': Specifies a list of directories where Django will search for templates. You can include the project-wide templates directory as well as additional directories for specific apps.

'APP_DIRS': Indicates whether Django should look for templates within each app's templates directory. When set to True, Django automatically searches within the templates directory of each installed app.

Custom Template Loaders:

You can also define custom template loaders if you need more advanced template loading behavior, such as loading templates from a database or remote location. Django provides flexibility to customize template loading to suit your project's needs.

Understanding how template loading works in Django allows you to organize your templates effectively and control how Django finds and loads them when rendering views.

2.6 TEMPLATE INHERITANCE

Template inheritance in Django allows you to create a base template with common elements and then extend it in child templates to override specific blocks or add additional content. This promotes code reusability and helps maintain a consistent layout across multiple pages. Let's walk through an example of template inheritance

2.6.1. Base Template

Create a base template that defines the overall structure of your website. This template typically includes the HTML structure, header, footer, and any common elements.

```
<!-- base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>{% block title %}My Website{% endblock %}</title>
```

```
</head>
<body>
  <header>
    <h1>Welcome to My Website</h1>
  </header>

  <nav>
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/about/">About</a></li>
      <li><a href="/contact/">Contact</a></li>
    </ul>
  </nav>

  <main>
    {% block content %}
    {% endblock %}
  </main>

  <footer>
    <p>&copy; 2024 My Website</p>
  </footer>
</body>
</html>
```

3.6.2 Child Template

Extend the base template in a child template and override specific blocks with content unique to that page.

```
<!-- home.html -->
{% extends 'base.html' %}


{% block title %}Home - My Website{% endblock %}


{% block content %}
<h2>Welcome to the Home Page</h2>
<p>This is the content of the home page.</p>
{% endblock %}
```

In this example:

The {% extends %} tag in the child template (home.html) specifies that it extends the base template (base.html).

The {% block %} tags define blocks within the base template that can be overridden by child templates.

In the child template (home.html), we override the title block with "Home - My Website" and the content block with content specific to the home page.

3.6.3 Render the Child Template:

Render the child template in a Django view as usual.

```
# views.py
from django.shortcuts import render
def home(request):
    return render(request, 'home.html')
```

output:

When a user visits the home page of your website, Django renders the home.html template, which extends the base.html template. The content defined in the home.html template is inserted into the {% block content %} block of the base.html template, resulting in a complete HTML page with the common layout provided by the base template.

This example demonstrates how to use template inheritance in Django to create reusable templates and maintain a consistent layout across multiple pages. You can extend this pattern to create additional child templates for different pages of your website.