MODULE-1: MVC BASED WEB DESIGNING

1.1 INTRODUCTION

What is Software?

Software refers to a collection of instructions, data, and programs that enable computers to perform specific tasks or functions. It's intangible and exists in the form of code written in programming languages that computers can understand and execute. Software is the interface between users and computer hardware, facilitating communication and enabling users to interact with computers to accomplish various objectives.

What is Software Architecture Design?

Software architecture design is the process of defining the structure, components, relationships, and behavior of a software system to meet specific requirements. It involves making high-level design decisions that determine how the system will be organized and how its components will interact with each other to achieve the desired functionality.

Components of software architecture design:

Component Identification: Identifying the major components or modules of the system based on functional requirements and domain knowledge. Components represent distinct units of functionality or logical groupings of related functionality.

Architectural Styles: Choosing an architectural style or pattern that best suits the requirements and constraints of the system. Common architectural styles include layered architecture, client-server architecture, microservices architecture, and event-driven architecture.

Decomposition and Abstraction: Breaking down the system into smaller, more manageable components through decomposition. This involves identifying subsystems, defining their boundaries, and establishing interfaces for communication between them. Abstraction is used to hide unnecessary details and focus on essential aspects of each component.

Communication Protocols: Defining communication protocols and data formats for interactions between system components. This includes specifying message formats, APIs, and protocols for communication over networks or between different layers of the system.

Data Management: Designing the data architecture of the system, including data storage, retrieval, and manipulation. This involves selecting appropriate data storage technologies (e.g., relational databases, NoSQL databases, file systems) and designing data models, schemas, and access patterns.

Scalability and Performance: Considering scalability and performance requirements during the design phase and incorporating design principles and patterns that support scalability

and optimize system performance. This may include techniques such as load balancing, caching, and asynchronous processing.

Security and Reliability: Incorporating security measures and reliability mechanisms to protect the system from security threats and ensure its resilience to failures. This includes implementing access controls, encryption, authentication, and error handling mechanisms.

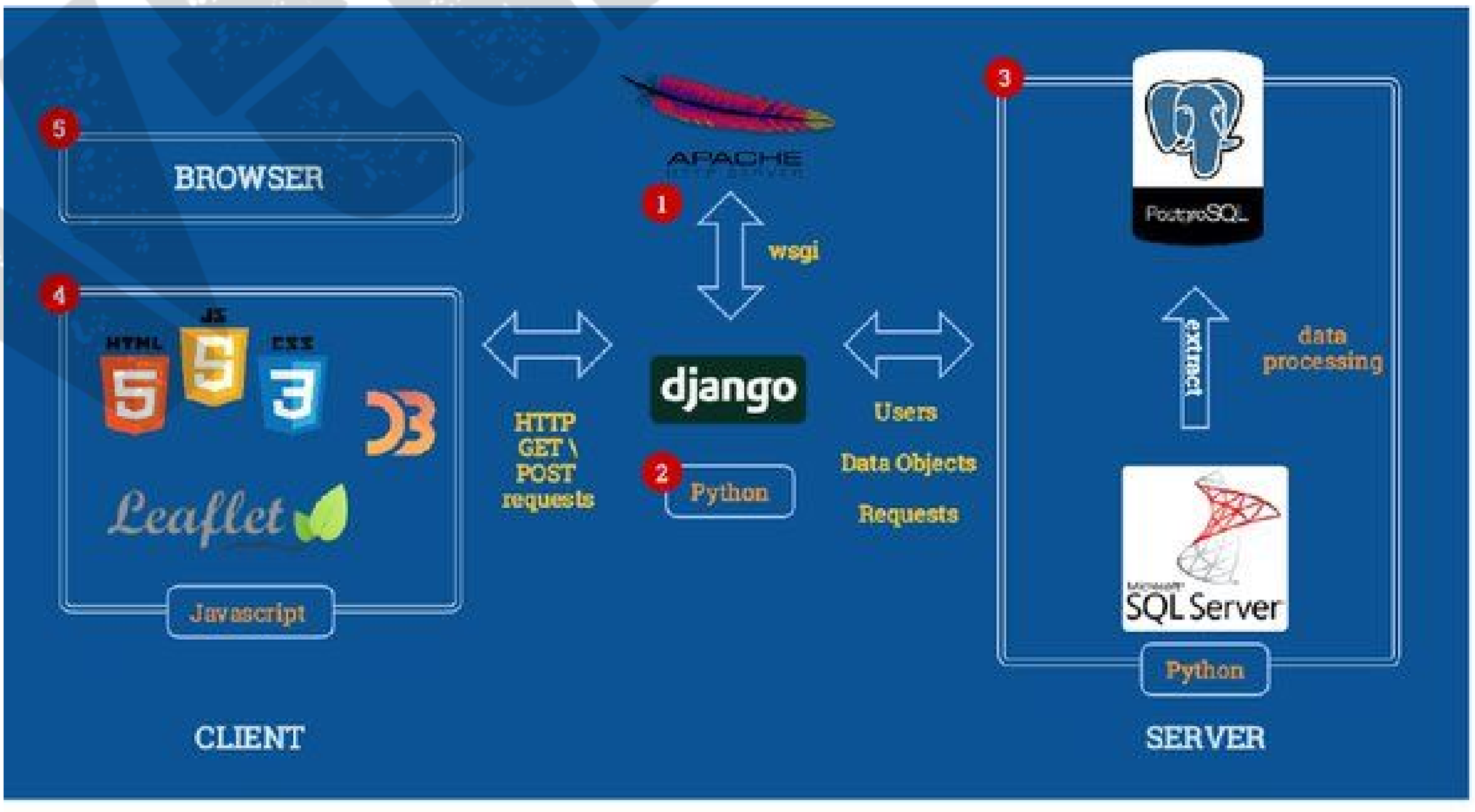
Cross-Cutting Concerns: Addressing cross-cutting concerns such as logging, monitoring, and configuration management that affect multiple components or aspects of the system. These concerns are typically addressed through reusable components or aspects of the architecture that are applied across the system.

Documentation and Governance: Documenting the architecture design decisions, rationale, and guidelines to ensure clarity and facilitate communication among stakeholders. Establishing governance processes to manage architectural changes and ensure adherence to architectural principles and standards.

Software architecture design is an iterative process that involves collaboration among architects, developers, stakeholders, and other relevant parties. It requires balancing conflicting goals and constraints, such as functionality, performance, scalability, and maintainability, to arrive at an architecture that meets the system's and its stakeholders' needs.

Explain Django full-stack development architecture.

Django full-stack development refers to using the Django web framework to develop both the front-end (client-side) and back-end (server-side) components of a web application. Django is a high-level Python web framework that enables the rapid development of secure, scalable, and maintainable web applications.



Back-end Development:

- **Model Layer**: Define data models using Django's Object-Relational Mapping (ORM) system. Models represent the structure of the application's data and interact with the database.
- **View Layer**: Implement views, which are Python functions or classes responsible for processing incoming HTTP requests, interacting with the database through models, and returning HTTP responses.
- **Controller (Business Logic):** Django's views act as controllers in the MVC pattern, handling the business logic of the application by processing requests, validating input, and generating responses.
- **URL Routing**: Define URL patterns in the URLconf (URL configuration) to map incoming URLs to views. Django's URL routing mechanism allows for clean and flexible URL patterns.
- **Middleware**: Implement middleware components to process requests and responses at various stages of the Django request/response cycle. Middleware can perform tasks such as authentication, session management, and error handling.

Front-end Development:

- **Templates**: Create HTML templates using Django's template engine, which allows for the dynamic generation of HTML content based on data provided by views.
- **Static Files Handling**: Serve static files (e.g., CSS, JavaScript, images) using Django's built-in static files handling capabilities. Static files are typically stored in the project's static directory.
- **Integration with Front-end Frameworks**: Optionally integrate Django with front-end frameworks like React.js, Vue.js, or Angular for building rich, interactive user interfaces. Django can serve as a RESTful API backend while the front-end framework handles the UI rendering and user interactions.

Database Interaction:

Django supports multiple databases including PostgreSQL, MySQL, SQLite, and Oracle. Developers can define database models using Django's ORM and perform database operations such as querying, inserting, updating, and deleting records.

Authentication and Authorization:

Implement user authentication and authorization using Django's built-in authentication system. Django provides tools for user management, password hashing, session management, and permissions handling out of the box.

Deployment and Scaling:

Deploy Django applications to production servers using platforms like Heroku, AWS, or DigitalOcean. Django applications can be scaled horizontally by adding more application instances behind a load balancer to handle increased traffic.

1.2 INSTALLATION

Open Terminal or Command Prompt:

Open a terminal or command prompt on your system.

Install Python (if not already installed):

If Python is not installed on your system, you need to install it first. You can download Python from the official website: python.org. Follow the installation instructions provided for your operating system.

Verify Python Installation:

After installing Python, verify that it's installed correctly by opening the terminal or command prompt and typing:

python --version

This command should display the installed Python version.

Virtual Environment

A virtual environment is a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages. It allows you to have multiple isolated Python environments on a single machine, each with its own Python interpreter and set of installed packages.

The purpose of using a virtual environment, especially when working with Django, includes:

Isolation: Virtual environments provide isolation between different projects and their dependencies. This means that each project can have its own set of dependencies without affecting other projects or the system-wide Python installation. This helps avoid conflicts between different versions of packages and ensures that each project remains self-contained and independent.

Dependency Management: By using a virtual environment, you can easily manage dependencies for your Django project. You can specify the exact versions of packages required for your project, ensuring that it works consistently across different environments and preventing unexpected issues due to dependency changes.

Reproducibility: Virtual environments make it easier to reproduce the development environment for your Django project on other machines. By sharing the project's

requirements.txt file (which lists all dependencies), other developers can quickly create the same environment using the same versions of packages.

Sandboxing: Virtual environments provide a sandboxed environment where you can install and experiment with different packages and versions without affecting the global Python installation or other projects. This allows for easy experimentation and testing of new libraries or features without risking the stability of other projects.

Deployment: When deploying your Django project to a production server, using a virtual environment ensures that the server environment matches your development environment closely. This helps minimize deployment issues and ensures that your project runs smoothly in the production environment.

To create and activate a virtual environment for a Django project, you can use Python's built-in venv module or third-party tools like virtualenv or pipenv. Once activated, you can install Django and other project dependencies within the virtual environment using pip.

Overall, using a virtual environment for Django development is considered a best practice and helps maintain a clean and organized development environment while ensuring consistency, reproducibility, and isolation of dependencies.

```
py -m venv myworld
```

This will set up a virtual environment, and create a folder named "myworld" with subfolders and files, like this:

Then you have to activate the environment, by typing this command:

```
myworld\Scripts\activate.bat
```

Install Django:

Now, you can use pip to install Django. In the terminal or command prompt, type:

```
pip install django
```

This command will download and install the latest version of Django from the Python Package Index (PyPI) along with its dependencies.

Verify Django Installation:

After installing Django, you can verify the installation by running the following command:

```
django-admin --version
```

This command should display the installed Django version.

Create a Django Project:

Now that Django is installed, you can create a new Django project. Navigate to the directory where you want to create the project and run:`django-admin startproject myproject`

Replace "myproject" with the name you want to give your project.

Run the Development Server:

Navigate into the newly created project directory:

```
cd myproject
```

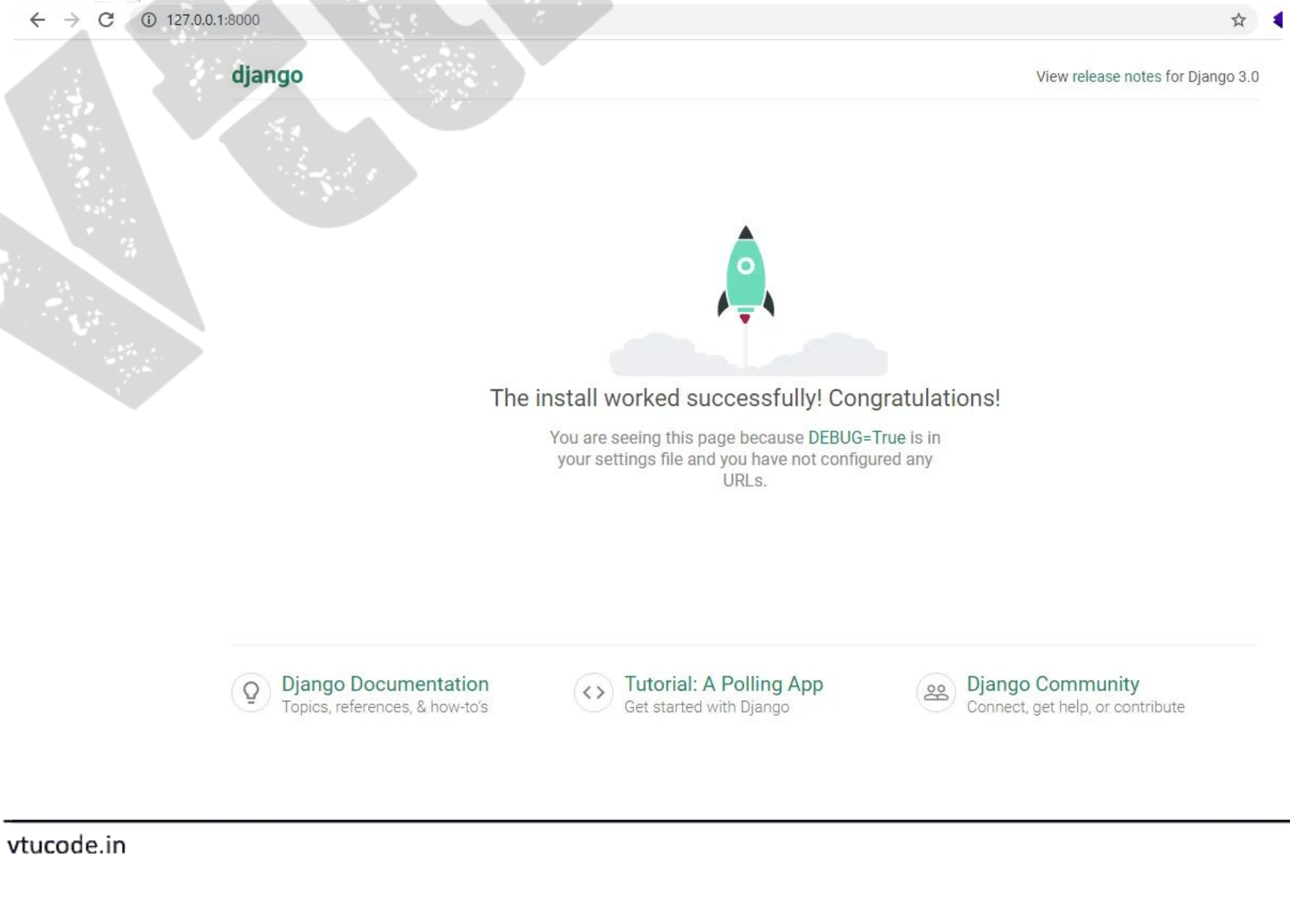
Then start the development server by running:

```
python manage.py runserver
```

This command will start the Django development server, and you should see output indicating that the server is running.

Access the Django Admin Interface:

Open a web browser and go to http://127.0.0.1:8000/admin/. You should see the Django admin login page.



That's it! You've now installed Django from scratch using pip and created a new Django project. You can continue developing your project by following the Django documentation and tutorials.

1.3 WEB FRAMEWORK

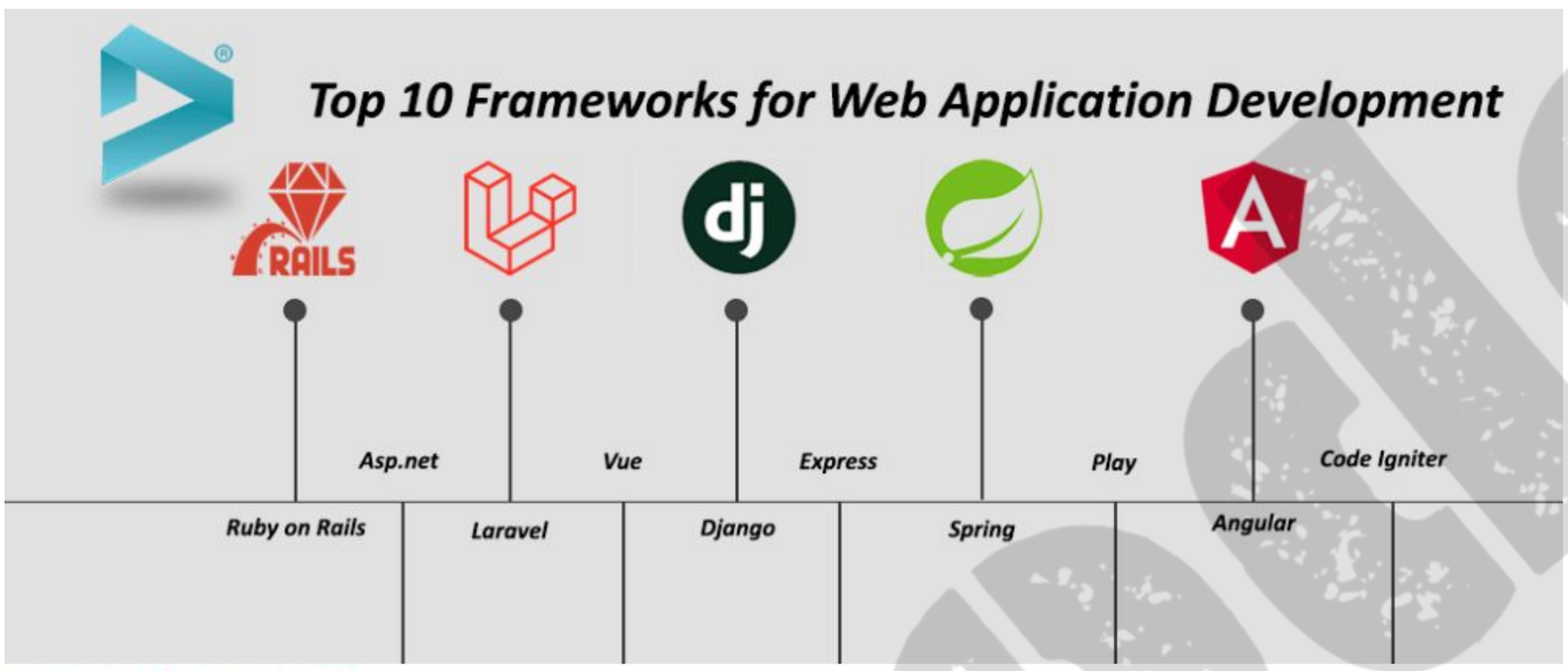
A web framework is a software framework that provides a structure and set of tools to facilitate the development of web applications. Web frameworks typically include libraries, modules, and pre-written code that abstract away common tasks, allowing developers to focus on building application-specific functionality rather than dealing with low-level details.

Key components and features of web frameworks:

- **Routing**: Web frameworks provide mechanisms for mapping URLs to specific code functions or classes, known as routes. This allows developers to define how different URLs should be handled by the application.
- **HTTP Request Handling**: Web frameworks handle incoming HTTP requests from clients (e.g., web browsers) and provide facilities for parsing request data, such as form submissions, query parameters, and request headers.
- **HTTP Response Generation**: Web frameworks facilitate the generation of HTTP responses to send back to clients. This includes rendering HTML templates, serializing data into various formats (e.g., JSON, XML), and setting response headers.
- **Template Engine**: Many web frameworks include a template engine that enables developers to generate dynamic HTML content by combining static HTML templates with dynamic data provided by the application.
- **Database Interaction**: Web frameworks often include libraries or modules for interacting with databases, such as ORM (Object-Relational Mapping) systems that map database tables to application objects and provide an abstraction layer for database operations.
- **Middleware**: Middleware components intercept HTTP requests and responses, allowing developers to perform common tasks such as authentication, logging, error handling, and request/response modification in a modular and reusable way.
- **Security Features**: Web frameworks provide features to help developers address common security concerns, such as protection against CSRF (Cross-Site Request Forgery) attacks, XSS (Cross-Site Scripting) prevention, input validation, and secure session management.
- **Session Management**: Web frameworks typically include mechanisms for managing user sessions, such as storing session data on the server or using client-side cookies to maintain session state.
- **Authentication and Authorization**: Many web frameworks include built-in support for user authentication and authorization, allowing developers to secure access to application resources based on user roles and permissions.

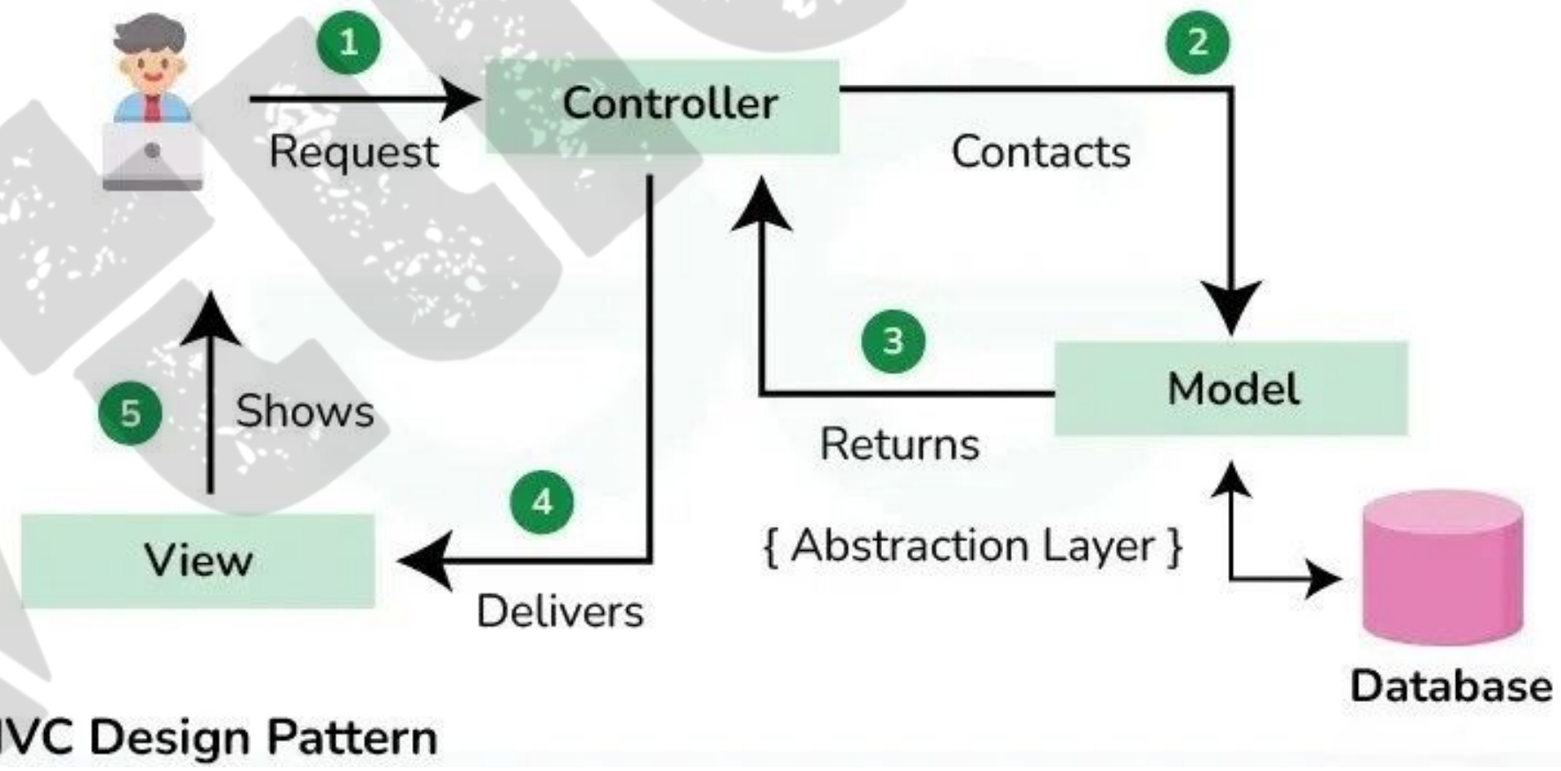
- **Testing Support**: Web frameworks often provide tools and utilities for writing and running automated tests to ensure the correctness and reliability of web applications.

Examples of popular web frameworks include Django (Python), Flask (Python), Ruby on Rails (Ruby), Express.js (Node.js), Laravel (PHP), Spring Boot (Java), and ASP.NET Core (C#). Each framework has its strengths, features, and ecosystem, catering to different programming languages and development preferences.



1.4 MVC DESIGN PATTERN

The Model-View-Controller (MVC) design pattern is a widely used architectural pattern for developing user interfaces in software applications. It divides the application into three interconnected components: Model, View, and Controller. Let's delve into the relationship and details of each component within the MVC architecture:



MVC Design Pattern

Model:

Responsibility: The Model component represents the application's data and business logic. It encapsulates the data structure and logic for manipulating that data.

Characteristics:

- It does not depend on the user interface or presentation layer.

- It typically interacts with the database, file system, web services, or other data sources to retrieve and manipulate data.
- It notifies the View component of any changes in the data (often through events or observers).

Example: In a web application, the Model might consist of classes representing entities like User, Product, or Order, along with logic for database operations like CRUD (Create, Read, Update, Delete).

View:

Responsibility: The View component is responsible for presenting the data to the user and gathering user input. It represents the user interface (UI) of the application.

Characteristics:

- It receives data from the Model and renders it in a format suitable for the user (e.g., HTML for web applications, GUI elements for desktop applications).
- It does not contain business logic; its primary role is to display data and handle user interactions.
- It may send user input (e.g., form submissions, button clicks) to the Controller for processing.

Example: In a web application, the View component comprises HTML templates, CSS stylesheets, and client-side scripts (e.g., JavaScript) responsible for rendering dynamic content and handling user interactions.

Controller:

Responsibility: The Controller component acts as an intermediary between the Model and the View. It receives user input from the View, processes it (possibly involving interactions with the Model), and updates the View accordingly.

Characteristics:

- It interprets user actions and translates them into operations on the Model.
- It updates the View based on changes in the Model and handles user interactions by invoking appropriate actions.
- It typically contains application logic related to routing, request handling, and business workflow orchestration.

Example: In a web application, the Controller component consists of server-side code (e.g., servlets, controllers in MVC frameworks like Spring MVC or Django) responsible for handling HTTP requests, interacting with the Model to perform business operations, and selecting the appropriate View to render.

Relationship:

The Model component interacts with the View indirectly through the Controller. It notifies the Controller of any changes in the data, which in turn updates the View.

The View component can send user input (e.g., form submissions, and button clicks) to the Controller for processing.

The Controller component communicates with both the Model and the View. It retrieves data from the Model, processes user input, and updates the View based on the changes in the Model.

Overall, the MVC pattern promotes the separation of concerns, making it easier to maintain, test, and evolve software applications by decoupling the user interface, data, and application logic.

1.5 DJANGO EVOLUTION

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

Version	Release Year	Key Features
0.9	2005	Initial public release
0.91	2006	Improved admin interface, new template filters
0.95	2006	Form library overhaul, enhanced documentation
0.96	2006	Session framework, generic views
1	2008	Stability guarantees, long-term support (LTS)
1.1	2009	Aggregates support, GeoDjango, improved testing
1.2	2010	Multiple database support, CSRF protection
1.3	2011	Class-based views, improved static files handling
1.4	2012	Time zone support, custom user model
1.5	2013	Custom template tags, configurable user model
1.6	2013	Improved testing tools, connection pooling

1.7	2014	Migrations framework, application loading refactor
1.8	2015	Built-in support for multiple template engines
1.9	2015	Automatic password validation, admin list filters
1.1	2016	Template-based widget rendering, conditional expressions
1.11	2017	Longer-term support, subquery expressions
2	2017	Python 3 only, new URL syntax, window expressions
2.1	2018	PostgreSQL 9.4+ JSONField support, easier testing
2.2	2019	Advanced options for database indexes, performance improvements
3	2019	ASGI support, MariaDB support, timezone-aware datetimes
3.1	2020	Support for customizing form rendering, asynchronous views
3.2	2021	Features new database backends, improved admin customization
4.2	2023	Psycopg 3 support, `ENGINE` as `django.db.backends.postgresql` supports both libraries.
5.0	2024	Django 5.0 supports Python 3.10, 3.11, and 3.12.

Key Features:

Django is a powerful web framework for building web applications quickly and efficiently. Some key features that make Django popular among developers:

Batteries-Included: Django follows the "batteries-included" philosophy, providing a comprehensive set of built-in features and tools for web development out of the box. This includes an ORM (Object-Relational Mapping) system for database interaction, a powerful admin interface, URL routing, authentication and authorization mechanisms, form handling, and more.

Model-View-Controller (MVC) Architecture: Django follows the Model-View-Controller (MVC) architectural pattern, although it's often described as a "Model-View-Template" (MVT) framework. This separation of concerns makes it easier to organize code, maintainability, and scale applications.

ORM (Object-Relational Mapping): Django's ORM simplifies database interactions by allowing developers to work with database models using Python objects. This abstraction layer handles database queries, transactions, and migrations, making it easier to manage database operations and maintain data consistency.

Admin Interface: Django provides a built-in admin interface that allows developers to create, read, update, and delete database records without writing custom admin views. The admin interface is highly customizable and can be tailored to suit specific application needs.

URL Routing: Django's URL routing system maps URLs to view functions or classes, allowing developers to define how different URLs should be handled by the application. This provides a clean and flexible way to organize application URLs and route requests to appropriate views.

Template Engine: Django's template engine enables developers to generate dynamic HTML content by combining static HTML templates with data provided by views. Templates support template inheritance, template tags, filters, and other features for building modular and reusable templates.

Security Features: Django prioritizes security and provides built-in protection against common web vulnerabilities such as CSRF (Cross-Site Request Forgery), XSS (Cross-Site Scripting), and SQL injection. Django's authentication system, authorization mechanisms, and session management features help developers implement secure user authentication and authorization.

ORM Migrations: Django's migration system automates the process of database schema changes, making it easier to evolve the database schema over time without manual intervention. Migrations track changes to database models and generate SQL scripts to apply those changes to the database.

Internationalization and Localization: Django supports internationalization (i18n) and localization (l10n) features, allowing developers to create multilingual web applications. Django provides tools for translating text strings, formatting dates, numbers, and handling language-specific content.

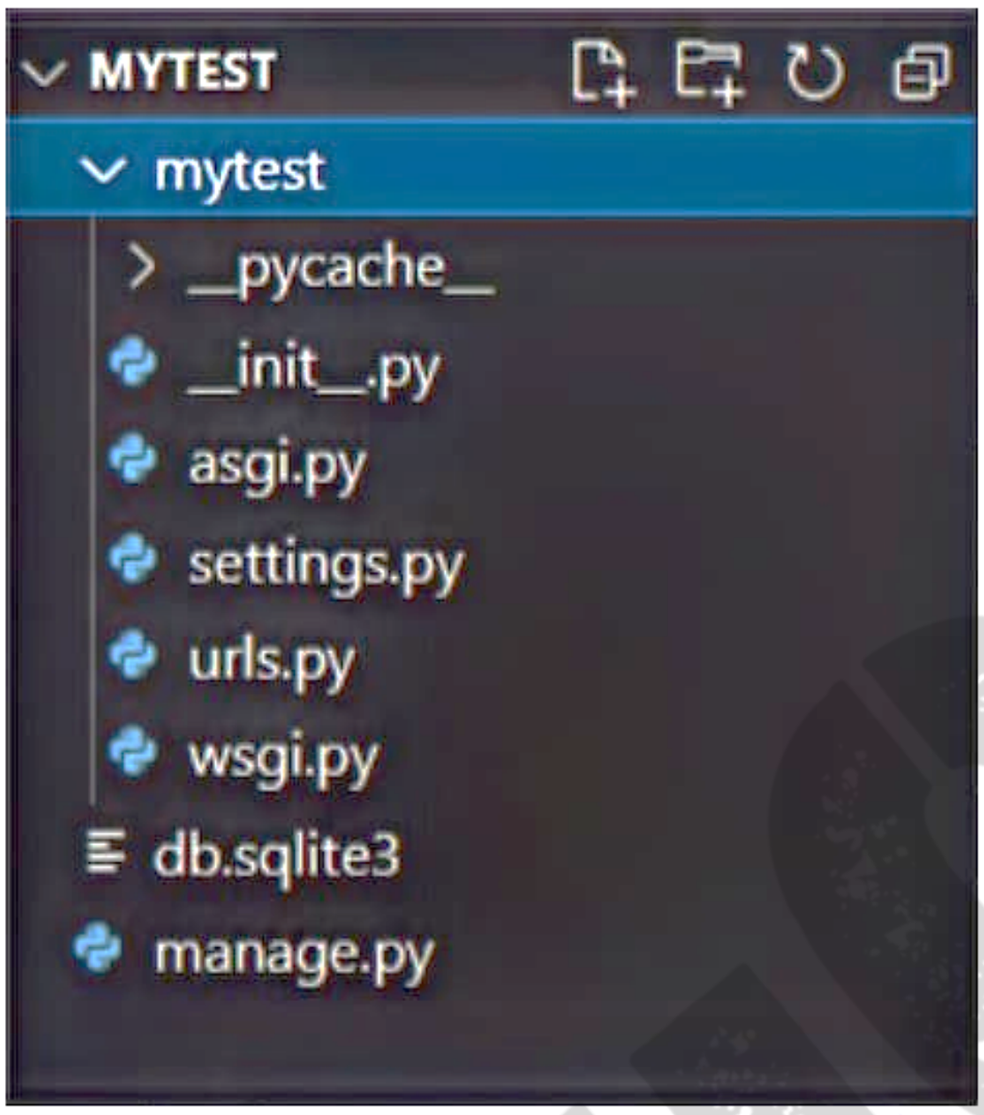
Scalability and Performance: Django is designed to scale well and can handle high traffic loads efficiently. It offers features like caching, session management, and support for distributed architectures to improve performance and scalability.

These key features, along with Django's emphasis on simplicity, flexibility, and productivity, make it a popular choice for building a wide range of web applications, from small personal projects to large-scale enterprise systems.

1.6 PROJECT STRUCTURE

django-admin startproject mytest

This will create a "myproject" folder with the following structure



In Django, project structure typically follows a specific convention, which helps maintain consistency and organization across projects. Here's a breakdown of the common structure:

- **Project Directory**: This is the main directory for your Django project. It contains everything related to your project.
- **Manage.py**: This is a command-line utility that lets you interact with your Django project. You can use it for various tasks like running a development server, creating database migrations, etc.
- **Settings.py**: This file contains all the configuration settings for your Django project. It includes settings like database configuration, static files configuration, middleware, installed apps, etc.
- **URLs.py**: This file contains URL patterns for your project. It maps URL paths to views.
- **Wsgi.py**: This file is the entry point for WSGI-compatible web servers("Web Server Gateway Interface.") to serve your Django application.
- **Asgi.py**: This file is similar to Wsgi.py but is used for ASGI-compatible("Asynchronous Server Gateway Interface.") web servers, which are used for asynchronous applications.

- **Apps**: Django applications are organized into individual apps. Each app typically represents a specific functionality of your project. Each app has its own models, views, URLs, and templates.
- **Models**: This directory contains Python classes that represent your data models. Each model typically maps to a database table.
- **Views**: Views contain the logic that processes a user's request and returns a response. They interact with models to retrieve or manipulate data and render templates to generate HTML responses.
- **URLs**: This file defines URL patterns specific to the app.
- **Templates**: Templates are HTML files that define the presentation layer of your application. They are typically used by views to generate dynamic content.
- **Static Files**: This directory contains static files like CSS, JavaScript, images, etc., which are served directly by the web server.
- **Templates**: Although each app can have its own templates directory, you may also have a global templates directory at the project level for templates that are shared across multiple apps.
- **Static Files**: Similarly, you may have a global static files directory for static files that are shared across multiple apps.
- **Migrations**: Django migrations are used to manage changes to your database schema. The migrations directory contains Python files that define these changes.
- **Static Root**: This is the directory where Django collects all the static files from different apps into a single location during deployment.
- **Media Root**: This is the directory where user-uploaded files are stored. It's configured in settings.py.

This structure provides a clear separation of concerns and makes it easier to manage and scale Django projects. Additionally, you can customize this structure to suit the specific needs of your project

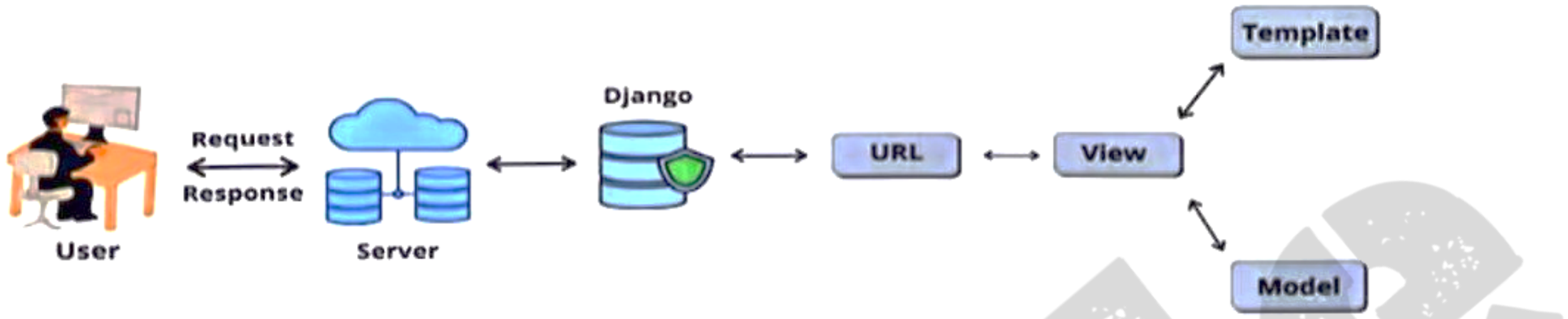
1.7 DJANGO ARCHITECTURE

Django adopts the MVT (Model-View-Template) architecture.

- **Model (M):** It represents the data layer, handling database interactions and logic.
- **View (V):** Views receive requests, process them, and generate responses. They contain the application's business logic.
- **Template (T):** Templates are HTML files with embedded Django Template Language (DTL) code for dynamic content rendering.

While MVT is akin to MVC (Model-View-Controller), Django differs in handling the controller part. Instead of using separate controllers, Django integrates controller-like functionality

into its templates. This means Django's templates, which combine HTML and DTL, take on some of the responsibilities traditionally attributed to controllers in MVC design patterns.



Model:

- Think of the Model as the data structure or the database of your web application.
- It represents the data and the logic to interact with the data.
- For example, if you're building a blog, your models might include classes like Post, Author, Comment, etc.

simple example of a Django model for a blog post:

```
from django.db import models

class Post(models.Model):

   title = models.CharField(max_length=100)

   content = models.TextField()

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

View:

- Views in Django are like the controllers in other frameworks, but with some differences.
- Views receive requests from users and return responses.
- They contain the logic to process requests and produce responses.
- Views interact with models to retrieve or manipulate data.

For example, a view to display a list of blog posts:

```
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})
```

Template:

- Templates are like HTML files but with additional Django Template Language (DTL) for dynamic content.
- They represent the presentation layer of your application.
- Templates render the data provided by views into HTML that's sent to the user's browser.

For example, a template to display a list of blog posts:

```
html
<!-- blog/post_list.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Blog Post List</title>
</head>
<body>
    <h1>Blog Posts</h1>
    <ul>
      {% for post in posts %}
        <li>{{ post.title }}</li>
      {% endfor %}
    </ul>
</body>
</html>
```

In summary, Django's MVT architecture separates the concerns of data (Model), logic (View), and presentation (Template) in a web application. Models handle data and database interactions, views process requests and generate responses, and templates render HTML

with dynamic content using Django's template language. This separation of concerns helps in building scalable and maintainable web applications.

Benefits of Django Framework:

- **Rapid Development**: With Django's design, developers can easily work on different parts of a website simultaneously. Imagine building a house where one team works on the foundation while another team works on the roof, speeding up the construction process.
- **Loosely Coupled**: In Django, the different components are interconnected only when needed, enhancing the security of the website. It's like storing valuables in a safe deposit box instead of leaving them out in the open.
- **Ease of Modification**: Django's architecture allows for changes in one part of the application without affecting other parts. It's similar to how you can rearrange furniture in one room without disturbing the layout of the entire house.
- **Security**: Django prioritizes security by protecting against common threats like click-jacking and SQL injections. It's like having a strong lock on your front door and security cameras to safeguard your home.
- **Scalability**: Django-powered websites can handle increasing numbers of users without sacrificing performance. Think of it like a stadium that can accommodate more and more fans without overcrowding or slowing down, just like Spotify, Netflix, and other popular sites built with Django.

1.8 DJANGO

Create virtual environment

pip install virtualenvwrapper-win

This command installs the virtualenvwrapper-win package using pip. Virtualenvwrapper-win is a set of extensions to Ian Bicking's virtualenv tool, providing commands to manage multiple virtual environments on Windows.

mkvirtualenv mylearn

This command creates a new virtual environment named "mylearn" using virtualenvwrapper-win. A virtual environment is an isolated Python environment where dependencies and packages can be installed without affecting the system-wide Python installation.

workon mylearn

This command activates the virtual environment named "mylearn". When a virtual environment is activated, any subsequent Python-related commands will use the packages and dependencies installed within that environment.

pip install Django

This command installs the Django web framework within the activated virtual environment. Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.

django-admin --version

This command checks the version of Django installed within the virtual environment. The django-admin command-line utility is used for various administrative tasks in Django projects, and the --version flag displays the installed Django version.

Creating A New Project In Django

To start a new Django project, run the command below:

1.django-admin startproject myproject

After we run the command above, it will generate the base folder structure for a Django project.

Right now, our myproject directory looks like this:

```
myproject/              <-- higher level folder
|-- myproject/          <-- django project folder
|   |-- myproject/
|   |   |-- __init__.py
|   |   |-- settings.py
|   |   |-- urls.py
|   |   |-- wsgi.py
|   +-- manage.py
```

manage.py: a shortcut to use the django-admin command-line utility. It's used to run management commands related to our project. We will use it to run the development server, run tests, create migrations, and much more.

__init__.py: this empty file tells Python that this folder is a Python package.

settings.py: this file contains all the project's configurations. We will refer to this file all the time!

urls.py: this file is responsible for mapping the routes and paths in our project. For example, if you want to show something in the URL /about/, you have to map it here first.

wsgi.py: this file is a simple gateway interface used for deployment.

Now everything is set up and we can run the server using the below command

2.python manage.py runserver

Now open the URL in a Web browser: http://127.0.0.1:8000 and you should see the success page, which means the server is running correctly.

Creating Django Apps : Django project contains many apps within it. An app can't run without a project. You can create an app by the following command

2. django-admin startapp articles

This will give us the following directory structure:

```
myproject/
 |-- myproject/
 |    |-- articles/          <-- our new django app!
 |    |    |-- migrations/
 |    |    |    +-- __init__.py
 |    |    |-- __init__.py
 |    |    |-- admin.py
 |    |    |-- apps.py
 |    |    |-- models.py
 |    |    |-- tests.py
 |    |    +-- views.py
 |    |-- myproject/
 |    |    |-- __init__.py
 |    |    |-- settings.py
 |    |    |-- urls.py
 |    |    |-- wsgi.py
 |    +-- manage.py
```

+-- venv

So, let's first explore what each file does:

migrations: here Django stores some files to keep track of the changes you create in the py file, and to keep the database and the models.py synchronized.

admin.py: this is a configuration file for a built-in Django app called Django Admin.

apps.py: this is a configuration file of the app itself.

models.py: here is where we define the entities of our Web application. The models are translated automatically by Django into database tables.

tests.py: this file is used to write unit tests for the app.

views.py: this is the file where we handle the request/response cycle of our Web application.

Now that we created our first app, let's configure our project to use it. Open settings.py file and find installed_apps block.

settings.py

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

Here we have added 'articles' in the installed apps section. Now the app is ready to run with the Django project.

Configuring Database : By default Django project comes with sqlite3 database but you can customize it to use MySQL or Postgresql as per the requirements.

settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'blog',
        'USER': 'postgres',
        'PASSWORD': '**********',
        'HOST': 'localhost',
        'PORT': 5432,
    }
}
```

Now your project is connected with Postgres database and ready to run.

So now, we have learned how the MVT pattern works and the structure of the Django application along with the database configuration.

Creating App in project in main myproject

python manage.py startapp calc

Running python manage.py startapp calc is a Django command used to create a new Django app named "calc" within your Django project. The details as below for the command:

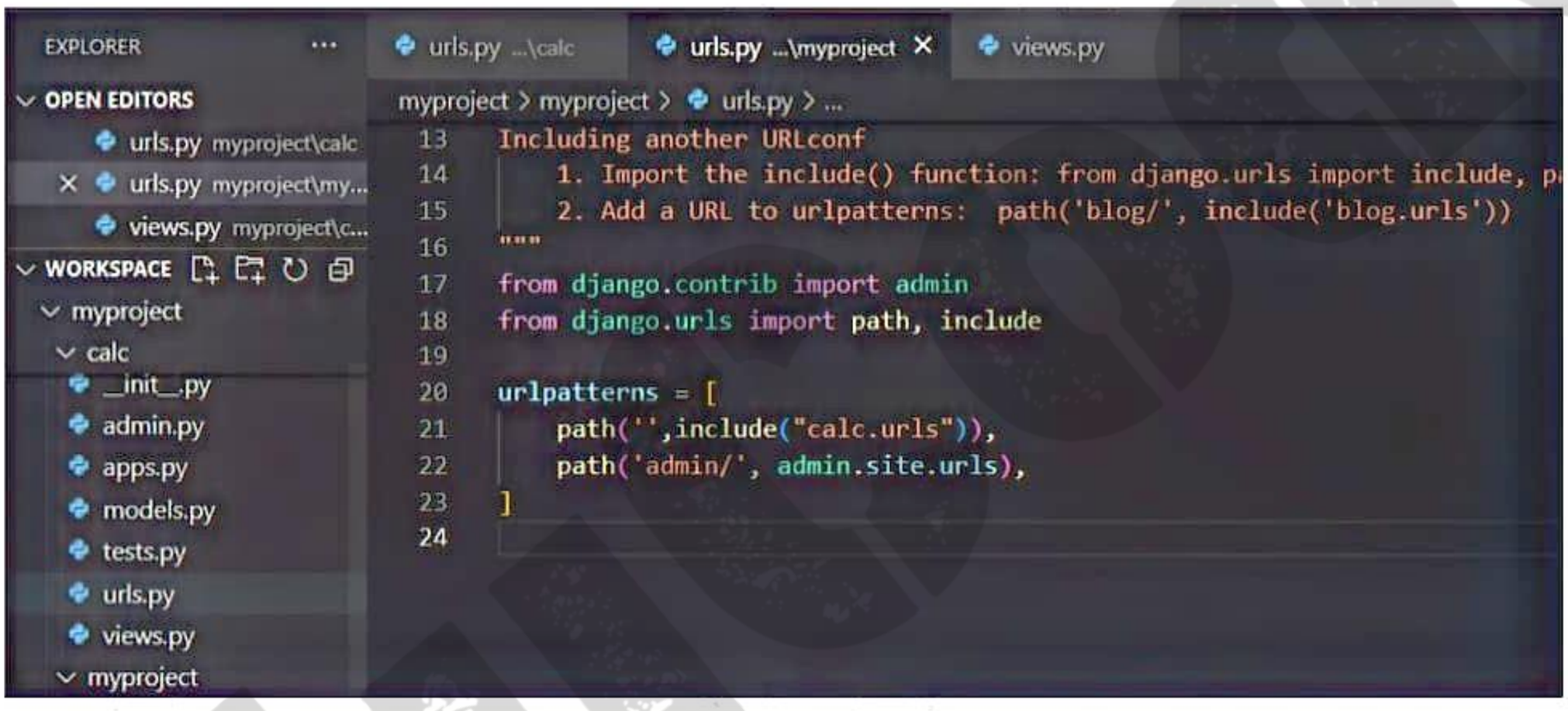
python: This is the Python interpreter used to execute the command.

manage.py: This is a script automatically created in the root directory of a Django project when you create a new project using django-admin startproject <project_name>. It's used to perform various administrative tasks related to the Django project.

startapp: This is a subcommand of manage.py used to create a new Django app within the project.

calc: This is the name of the Django app being created. In this case, it's named "calc", but you can replace it with any name you prefer for your app.

When you run ***python manage.py startapp calc***, Django will generate a directory structure and files for your new app within your project directory. This includes files for models, views, templates, and other components typical of a Django app. You'll find the newly created "calc" directory inside your project's directory.



create urls.py

```
from django.urls import path

from . import views


urlpatterns = [
    path('',views.home ,name='home'),

]
```

This is a snippet from a Django project's urls.py file, where URL patterns for routing within the Django application are defined. Following are the details of each statement

from django.urls import path: This line imports the path function from Django's urls module. The path function is used to define URL patterns for routing within a Django application.

from . import views: This line imports the views module from the current directory (indicated by .). In Django, views are Python functions or classes that handle HTTP requests and return HTTP responses.

urlpatterns: This is a list that contains URL patterns for routing within the Django application.

path('', views.home, name='home'): This line defines a URL pattern. It consists of three parts:

- The first argument '' represents the URL pattern itself. In this case, it's an empty string, indicating that this URL pattern corresponds to the root URL of the application.
- The second argument views.home refers to the home function (or view) imported from the views module. This specifies the view that should be called when the corresponding URL pattern is matched.
- The third argument name='home' is an optional argument that provides a unique name to the URL pattern. This is useful for referring to the URL pattern in Django templates or in other parts of the codebase. In this case, the name 'home' is assigned to this URL pattern.

In summary, this snippet defines a single URL pattern that corresponds to the root URL of the Django application. When a request is made to the root URL, it will be routed to the home view function defined in the views.py module. The URL pattern is named 'home' for easy reference in the Django project.

Views.py

```
from django.shortcuts import render

from django.http import HttpResponse


# Create your vie:ws here.

def home(request):
    return HttpResponse("WELCOME TO HELLO WORLD")
```

This is a snippet from a Django application's views.py file, which contains Python functions or classes that handle HTTP requests and return HTTP responses.

from django.shortcuts import render: This line imports the render function from Django's shortcuts module. The render function is used to render HTML templates with context data and return an HttpResponse object.

from django.http import HttpResponse: This line imports the HttpResponse class from Django's http module. The HttpResponse class is used to construct HTTP responses.

def home(request): This line defines a Python function named home that takes a request object as its argument. In Django, view functions take a request object as input, which contains information about the HTTP request made by the client.

return HttpResponse("WELCOME TO HELLO WORLD"): This line returns an HttpResponse object with the string "WELCOME TO HELLO WORLD" as its content. This means that when the home view function is called in response to an HTTP request, it will return a simple HTTP response with the specified content.

In summary, this snippet defines a single view function named home that returns a simple HTTP response with the text "WELCOME TO HELLO WORLD". This view function will be called when the corresponding URL pattern (defined in the urls.py file) is matched, and it will generate the HTTP response that is sent back to the client.

Main project:

Urls.py should be modified

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


urlpatterns = [
    path('',include("calc.urls")),
    path('admin/', admin.site.urls),
]
```

This snippet is from the main urls.py file in a Django project, which serves as the central routing configuration for the entire project. Here's an explanation of each part:

from django.contrib import admin: This line imports the admin module from Django's contrib package, which provides the Django admin interface.

from django.urls import path, include: This line imports the path function and the include function from Django's urls module. The path function is used to define URL patterns, while the include function is used to include URL patterns from other Django apps.

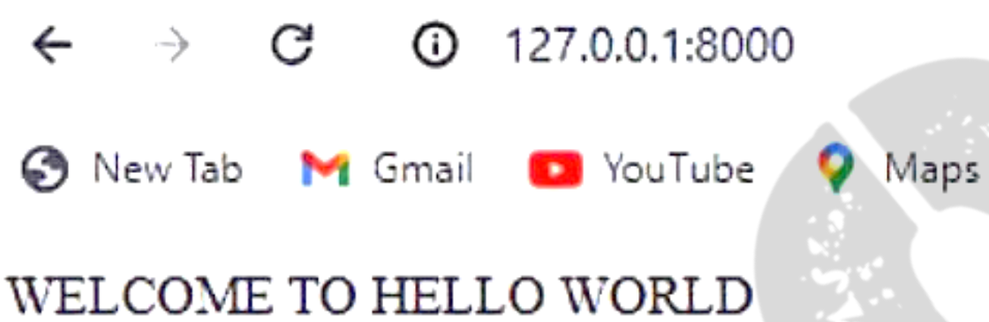
urlpatterns: This is a list that contains URL patterns for routing within the Django project.

path('', include("calc.urls")): This line includes URL patterns defined in the urls.py file of the "calc" Django app. The empty string '' indicates that these URL patterns will be included at the root level of the project. So, any URL matching the root will be directed to the URLs defined in the "calc" app's urls.py file.

path('admin/', admin.site.urls): This line maps the URL path 'admin/' to the Django admin interface. When a user visits /admin/ in the browser, they will be directed to the Django admin interface. The admin.site.urls function includes the admin URLs.

In summary, this urls.py configuration includes the URL patterns defined in the "calc" app's urls.py file at the root level of the project and also maps the /admin/ URL to the Django admin interface.

Output:



1.9 VIEWS

In Django, views are Python functions or classes that receive web requests and return web responses. They encapsulate the logic for processing HTTP requests and generating appropriate responses. Here are some details about views in Django along with examples:

1.9.1 Function-based Views (FBVs)

Function-based views are simple Python functions that take an HTTP request as input and return an HTTP response.

They are defined in views.py file within Django apps.

an example of a function-based view:

```
from django.http import HttpResponse

def my_view(request):
    # Process the request
    return HttpResponse("Hello, World!")
```

Explaination:

```
from django.http import HttpResponse
```

Line 1: This imports the HttpResponse class from the django.http module. HttpResponse is a class used to construct HTTP responses in Django.

```
def my_view(request):
```

Line 2: This defines a Python function named my_view that takes a request object as its argument. In Django, views are typically implemented as functions or class-based views. The request object contains information about the incoming HTTP request.

```
    # Process the request
```

Line 3: This is a comment line indicating that the subsequent code processes the request. You would typically put your logic for processing the request here.

```
    return HttpResponse("Hello, World!")
```

Line 4: This line constructs an HttpResponse object with the content "Hello, World!". This response will be sent back to the client who made the request. In this case, the response simply contains the string "Hello, World!".

In summary, the provided code defines a view function named my_view that processes incoming HTTP requests and returns an HTTP response with the content "Hello, World!". This is a basic example of how to implement a simple function-based view in Django.

1.9.2 Class-based Views (CBVs)

Class-based views are views implemented as Python classes.

They provide an object-oriented way to define views, making it easier to reuse and organize code.

Django provides various generic class-based views for common tasks.

an example of a class-based view:

```
from django.views import View

from django.http import HttpResponse

class MyView(View):
    def get(self, request):
        # Process GET request
        return HttpResponse("Hello, World!")
```

This code snippet defines a class-based view (CBV) named MyView using Django's class-based views framework. Here's a breakdown of each part:

from django.views import View

from django.http import HttpResponse

Line 1-2: These lines import the View class from django.views module and the HttpResponse class from django.http module. View is a base class for all class-based views in Django, and HttpResponse is a class used to construct HTTP responses.

class MyView(View):

Line 3: This line defines a new class named MyView that inherits from the View class. This means that MyView is a subclass of View, and it inherits all the methods and attributes of the View class.

def get(self, request):

Line 4: This line defines a method named get within the MyView class. In Django class-based views, methods like get, post, put, etc., correspond to HTTP request methods. In this case, get method handles HTTP GET requests.

Process GET request

Line 5: This is a comment indicating that the subsequent code processes the GET request. This is where you would typically put your logic for handling the GET request.

return HttpResponse("Hello, World!")

Line 6: This line constructs an HttpResponse object with the content "Hello, World!". This response will be sent back to the client who made the GET request. In other words, when a GET request is made to the MyView, it will respond with "Hello, World!".

In summary, the provided code defines a class-based view MyView that handles HTTP GET requests. When a GET request is made to this view, it responds with an HTTP response containing the string "Hello, World!". This is a basic example of how to implement a class-based view in Django to handle GET requests

Modified in Urls.py(in myproject)

from django.urls import path

from .views import home # Import the home view

urlpatterns = [

path('',home.as_view() ,name='home'), # Use home.as_view() for the view

]

This code snippet is from a Django project's urls.py file, where URL patterns for routing within the Django application are defined.

from django.urls import path

Line 1: This line imports the path function from Django's urls module. The path function is used to define URL patterns for routing within a Django application.

from .views import home

Line 2: This line imports the home view from the views module in the current directory (indicated by .). This assumes that there's a module named views.py in the same directory as the urls.py file, and it contains a view named home.

```
urlpatterns = [
    path('', home.as_view(), name='home'),
]
```

Line 3-5: This block defines the urlpatterns list, which contains URL patterns for routing within the Django application.

path('', home.as_view(), name='home'): This line defines a URL pattern.

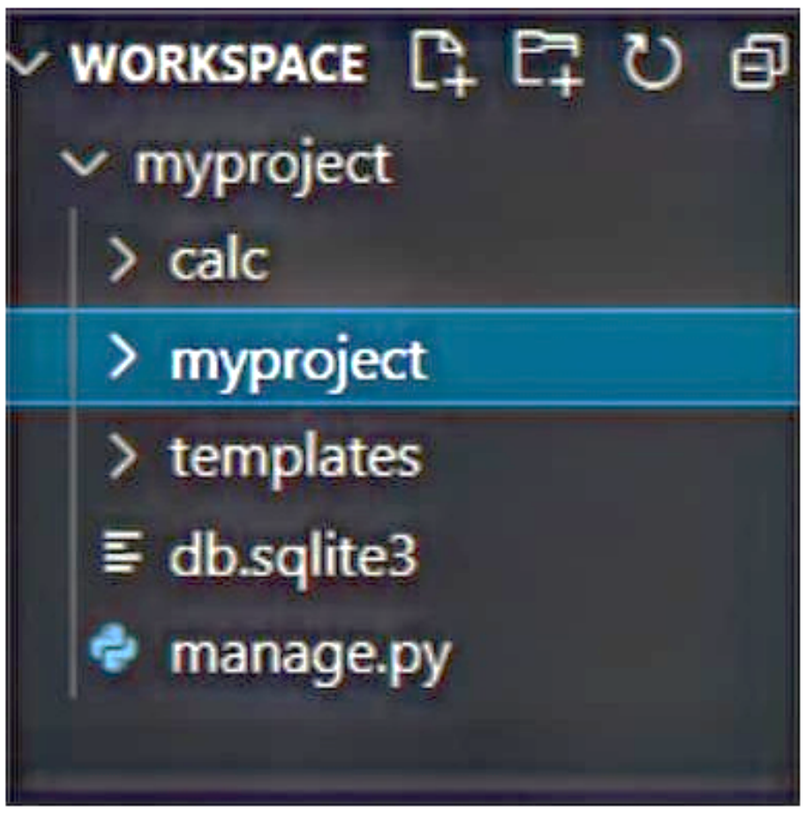
- *The first argument '' represents the URL pattern itself. In this case, it's an empty string, indicating that this URL pattern corresponds to the root URL of the application.*
- *The second argument home.as_view() refers to the home view imported earlier. Since home is a class-based view, .as_view() method is used to convert it into a view callable that Django's URL routing system can use.*
- *The third argument name='home' is an optional argument that provides a unique name to the URL pattern. This is useful for referring to the URL pattern in Django templates or in other parts of the codebase. In this case, the name 'home' is assigned to this URL pattern.*

In summary, this urls.py configuration sets up a single URL pattern that corresponds to the root URL of the Django application. When a request is made to the root URL, it will be routed to the home view, which is converted to a view callable using .as_view(). The URL pattern is named 'home' for easy reference in the Django project.

1.9.3 Rendering Templates:

Rendering templates in Django involves using the render() function provided by django.shortcuts to generate HTML content dynamically. This HTML content is typically based on a template file that contains HTML markup along with template language syntax for inserting dynamic data.

Here's a step-by-step explanation of how to render templates in Django:



Create a Template File:

First, create a template file (usually with a .html extension) in the templates directory of your Django app or project. You can organize templates into subdirectories within the templates directory if needed.

Example **template file (my_template.html):**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My Template</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

Define a View Function:

Next, define a view function in your Django app's views.py file. This view function will use the render() function to render the template with dynamic data.

Example view function:

```
from django.shortcuts import render
def home(request):
    context = {'name': 'Mallikarjuna'}
    return render(request, 'my_template.html', context)
```

Render the Template in the View:

Within the view function, use the render() function to render the template. Pass the request object, the name of the template file ('my_template.html' in this case), and a dictionary containing any dynamic data (context) to be passed to the template.

Example usage of render():

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

Access Dynamic Data in the Template:

In the template file, you can access the dynamic data passed from the view using template language syntax (usually Django's template tags).

Example usage of dynamic data in the template:

```
<h1>Hello, {{ name }}!</h1>
```

Setting.py

```
import os
' DIRS': [os.path.join(BASE_DIR,'templates')]
```

You can use the os.path.join() function to construct file paths in a platform-independent way. This function takes multiple path components as arguments and joins them together using the appropriate path separator for the current operating system.

os.path.join(BASE_DIR, 'templates')

In this example, BASE_DIR is assumed to be a variable containing the base directory of your Django project. By joining BASE_DIR with 'templates', you create a file path that points to the templates directory within your project.

The constructed file path can be used in various settings in your Django project, such as TEMPLATES for specifying the directories where Django should look for templates.

```
TEMPLATES = [
    {
```

```
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
    # Other template settings...
  },
]
```

In this example, the DIRS option specifies a list of directories where Django should look for template files. By including the file path constructed using os.path.join(), you ensure that Django will search for templates in the templates directory of your project.

URL Configuration:

Finally, configure the URL patterns in your Django project's urls.py file to map a URL path to the view function.

Example URL configuration:

```
from django.urls import path
from .views import home
urlpatterns = [
  path('',home ,name='home'),
]
```

With these steps, when a user accesses the URL path /my-url/ in the browser, Django will execute the my_view function, render the my_template.html template with the provided context, and return the resulting HTML content as an HTTP response.

1.9.4 Handling Form Submissions

Views are commonly used to handle form submissions.

They process form data and perform actions such as saving data to the database.

Here's an example of handling form submissions:

```
from django.http import HttpResponse
from django.shortcuts import render
from .forms import MyForm
def my_form_view(request):
  if request.method == 'POST':
```

```
    form = MyForm(request.POST)
    if form.is_valid():
        # Process the form data
        return HttpResponse("Form submitted successfully!")
else:
    form = MyForm()
return render(request, 'my_form_template.html', {'form': form})
```

Authentication and Authorization:

Views are used to implement user authentication and authorization logic.

They can restrict access to certain views based on user permissions or roles.

Here's an example of using built-in decorators for authentication:

```
from django.contrib.auth.decorators import login_required
from django.http import HttpResponse
@login_required
def my_protected_view(request):
    return HttpResponse("You're logged in!")
```

These examples illustrate the versatility and importance of views in Django for handling various aspects of web development, including rendering dynamic content, processing form submissions, and managing user authentication.

1.10 MAPPING URL TO VIEWS

Mapping URLs to views in Django involves defining URL patterns in the URL configuration (urls.py) of your Django project or app. These URL patterns specify which views should be invoked when a particular URL is accessed. Here's how to map URLs to views in Django:

Create a View Function or Class:

First, define a view function or class in your Django app's views.py file. This view will contain the logic for handling the HTTP request and generating the HTTP response.

Example view function:

```
from django.http import HttpResponse
def my_view(request):
```

```
    return HttpResponse("Hello, World!")
```

Example class-based view:

```
from django.views import View
from django.http import HttpResponse
class MyView(View):
    def get(self, request):
        return HttpResponse("Hello, World!")
```

Define URL Patterns:

Next, define URL patterns in your Django project's or app's urls.py file. Each URL pattern is a mapping between a URL pattern (expressed as a regular expression) and a view function or class.

Example URL configuration (urls.py):

```
from django.urls import path
from .views import my_view, MyView
urlpatterns = [
    path('hello/', my_view, name='hello'),
    path('hello-class/', MyView.as_view(), name='hello-class'),
]
```

Include App URLs in Project URLs (if applicable):

If you're defining URLs in an app, make sure to include the app's URLs in the project's URL configuration (urls.py). This is done using the include() function from django.urls.Example project URL configuration (urls.py):

```
from django.urls import path, include
urlpatterns = [
    path('myapp/', include('myapp.urls')),
    # Other URL patterns...
]
```

Access Views via URLs:

Once the URL patterns are defined, you can access the views by navigating to the corresponding URLs in your web browser or by making HTTP requests to those URLs programmatically.

For the view function my_view, you can access it at /hello/ URL.

For the class-based view MyView, you can access it at /hello-class/ URL.

By mapping URLs to views in this way, you can define the structure of your Django project's URL routing and specify which views should handle incoming HTTP requests for different URLs.

1.11 WORKING OF DJANGO URL CONFS AND LOOSE COUPLING

1.11.1 URLconfs in Django:

In Django, URLconfs serve as configuration files where you define the mapping between URLs and views within your project. They allow you to specify how incoming HTTP requests should be routed to the appropriate view functions or classes for processing.

URLconfs are Python modules that typically reside in the urls.py file of each Django app or in the project-level urls.py file. You can have multiple URLconfs, each handling a different set of URL patterns.

URLconfs contain a list of URL patterns defined using Django's urlpatterns variable. Each URL pattern is defined using the path() or re_path() function, which takes a URL pattern (as a string or regular expression) and a view function or class as arguments.

When Django receives an incoming HTTP request, it traverses through the urlpatterns list in each URLconf, attempting to match the requested URL against the defined patterns. If a match is found, Django invokes the corresponding view function or class to handle the request.

1.11.2. Loose Coupling in Django URLconfs:

- **Decoupling of URLs and Views**: Django's URLconfs promote loose coupling between URLs and views, which means they are not tightly interconnected. This separation allows you to modify the URL structure of your application without impacting the underlying view logic, and vice versa.
- **Modularity**: URLconfs enable you to organize your project's URL routing in a modular and reusable manner. Each app in your Django project can have its own URLconf, encapsulating the URL patterns specific to that app. This modular approach enhances code organization and maintainability.
- **Flexibility**: By decoupling URLs from views, Django allows you to reuse views across multiple URL patterns or even across different apps within your project. This flexibility facilitates code reuse and promotes the development of modular, composable components.

- **Scalability**: Loose coupling provided by URLconfs makes it easier to scale and extend your Django project over time. As your project grows, you can add new URL patterns and views without tightly coupling them to existing components, thus avoiding unnecessary dependencies and complexities.

Example:

Consider a scenario where you have an e-commerce Django application with multiple apps such as products, orders, and cart. Each app has its own URLconf (urls.py) where you define URL patterns specific to that app. For example:

products/urls.py:

```
from django.urls import path
from . import views
urlpatterns = [
    path('products/', views.ProductListView.as_view(), name='product-list'),
    path('products/<int:pk>/', views.ProductDetailView.as_view(), name='product-detail'),
]
```

orders/urls.py:

```
from django.urls import path
from . import views
urlpatterns = [
    path('orders/', views.OrderListView.as_view(), name='order-list'),
    path('orders/<int:pk>/', views.OrderDetailView.as_view(), name='order-detail'),
]
```

In this example, each app encapsulates its own URL patterns and view logic, promoting loose coupling between URLs and views and making the application more modular, maintainable, and scalable.

In summary, Django's URLconfs play a crucial role in facilitating loose coupling between URLs and views in your project. By separating URL routing from view logic, URLconfs promote modularity, flexibility, and scalability, making it easier to develop, maintain, and extend Django applications.

1.12 ERRORS IN DJANGO, WILD CARD PATTERNS IN URLS

In Django, wild card patterns in URLs are implemented using regular expressions, specifically with the re_path() function or the path() function with a route converter. These wild card patterns allow you to capture dynamic parts of a URL and pass them as parameters to your views. However, errors can occur when defining or using these patterns. Let's explore common errors and how to address them:

1. Incorrect Regular Expression:

If the regular expression used in the URL pattern is incorrect or invalid, Django will raise an error when processing the URLconf.

Double-check the regular expression syntax and ensure it matches the intended URL pattern. Common mistakes include missing parentheses, incorrect escape characters, or invalid quantifiers.

Error: Suppose you intend to match URLs like /articles/2022/ using a regular expression, but you mistakenly use an invalid syntax in the URL pattern definition.

Example:

```
# Incorrect URL pattern with invalid regular expression syntax

from django.urls import re_path
from . import views

urlpatterns = [
    re_path(r'^articles/\d{4}/$', views.article_detail_view),
]
```

Solution: Ensure the regular expression syntax is correct. In this case, the correct syntax for matching four-digit numbers would be \d{4} to match exactly four digit

2. Conflicting URL Patterns:

If you define multiple URL patterns that match the same URL, Django won't know which view to invoke, resulting in a URLResolverMatch error.

Review your URLconf and ensure that each URL pattern is unique. If necessary, reorganize your URL patterns or use more specific regular expressions to avoid conflicts.

Error: You define two URL patterns that match the same URL, causing a conflict.

Example:

```
# Conflicting URL patterns
```

```
urlpatterns = [
    path('articles/<int:year>/', views.year_archive),
    path('articles/<str:slug>/', views.article_detail),
]
```

Solution: Review your URL patterns and ensure each one is unique. You might need to use more specific regular expressions or reorder your patterns to avoid conflicts.

3. Missing Route Converter:

When using the path() function with route converters, forgetting to specify a converter for a dynamic part of the URL can lead to errors.

Ensure that each dynamic part of the URL defined in the path() function has a corresponding route converter specified. Common converters include <int:variable_name> for integers, <str:variable_name> for strings, etc.

Error: Forgetting to specify a route converter for a dynamic part of the URL in the path() function can lead to errors.

Example:

```
# Missing route converter
urlpatterns = [
    path('articles/<year>/', views.article_detail),  # Missing route converter for 'year'
]
```

Solution: Ensure that each dynamic part of the URL has a corresponding route converter specified. For example, <int:year> for integers.

4. Unmatched URL Parameters:

If your view function expects URL parameters but doesn't receive them due to mismatched URL patterns, Django will raise an error indicating missing arguments.

Ensure that your view function signature matches the URL pattern defined in the URLconf. If the view expects parameters, make sure the corresponding URL pattern captures and passes those parameters.

Error: Your view function expects URL parameters but doesn't receive them due to mismatched URL patterns.

Example:

```
# View expects parameters but doesn't receive them

def article_detail(request, year, month, day):

    ...
```

Solution: Make sure the URL pattern captures and passes the required parameters to the view function. For example, path('articles/<int:year>/<int:month>/<int:day>/', views.article_detail).

5. Invalid View Function:

If the view function specified in the URL pattern does not exist or is incorrectly referenced, Django will raise an error indicating that the view is not callable.

Double-check the view function name and import statement in your URLconf. Ensure that the view function is correctly defined and imported from the appropriate module.

Error: The view function specified in the URL pattern does not exist or is incorrectly referenced.

Example:

```
# Incorrect view function name or reference

urlpatterns = [

    path('articles/<int:year>/', views.non_existent_view),  # Incorrect view function name

]
```

Solution: Verify the view function name and import statement in your URLconf. Ensure the view function is correctly defined and imported.

6. Incorrect URL Reverse Lookup:

When using Django's reverse() function to generate URLs based on view names, errors can occur if the view name is misspelled or not defined.

Check your usage of the reverse() function and verify that the view name provided matches the name defined in your URLconf. Ensure that the view name is spelled correctly and consistently.

Error: Errors occur when using Django's reverse() function to generate URLs based on view names.

Example:

Incorrect reverse lookup

from django.urls import reverse

url = reverse('incorrect_view_name') # Incorrect view name

Solution: Double-check the view name passed to the reverse() function. Ensure it matches the name defined in your URLconf and is spelled correctly

7. Missing Import Statements:

For class-based views or views defined in separate modules, forgetting to import the view in your URLconf can lead to errors when referencing the view.

Double-check your import statements in the URLconf and ensure that all views referenced in URL patterns are correctly imported from their respective modules.

By addressing these common errors and ensuring the correct usage of wild card patterns in URLs, you can avoid issues and build robust URL routing in your Django application.

Error: Forgetting to import the view function or class in your URLconf can lead to errors when referencing the view.

Example:

Missing import statement for view function

urlpatterns = [

  path('articles/<int:year>/', non_existent_views.article_detail), # Missing import for 'non_existent_views'

]

Solution: Ensure all views referenced in URL patterns are correctly imported from their respective modules.

By addressing these common errors and ensuring the correct usage of wild card patterns in URLs, you can avoid issues and build robust URL routing in your Django application.