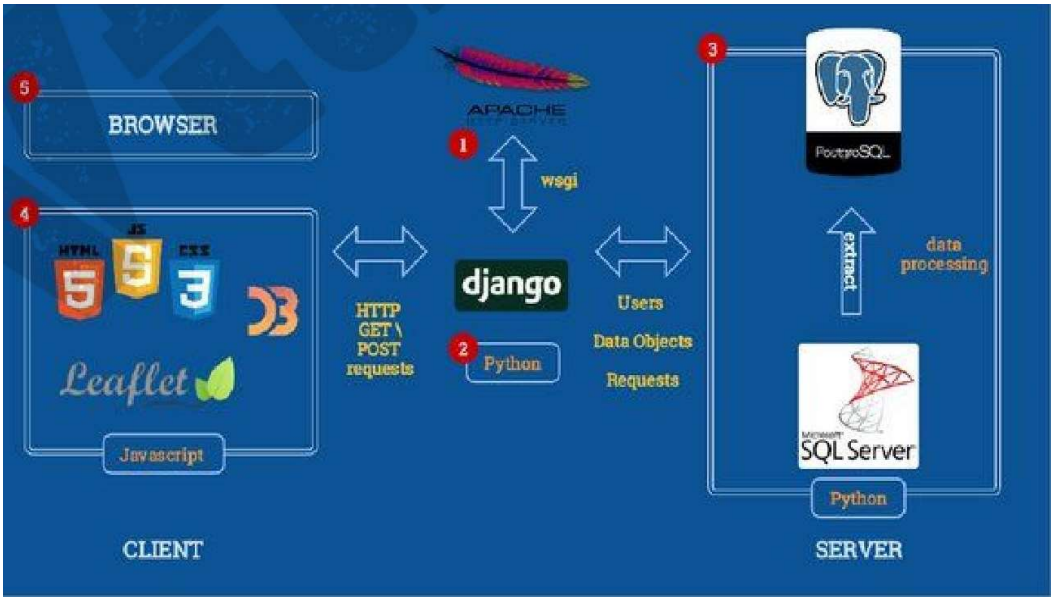
Web framework, MVC Design Pattern, Django Evolution, Views, Mapping URL to Views, Working of Django URL Confs and Loose Coupling, Errors in Django, Wild Card patterns in URLS.

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.

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.



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 selfcontained 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.

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.

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.

Install Django:

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

pip install django

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

Run the Development Server:

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.O.O.1:8000/admin/. You should see the Django admin login page.

Web Framework

web application frameworks' or 'web frameworks' as "a software framework that is designed to support the development of web applications including web services, web resources and web APIs". In simple words, web frameworks are a piece of software that offers a way to create and run web applications. Thus, you don't need to code on your own and look for probable miscalculations and faults.

Django is a Python-based web application framework that is free and open source. A framework is simply a collection of modules that facilitate development. They're grouped together and allow you to build apps or websites from scratch rather than starting from scratch.

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.

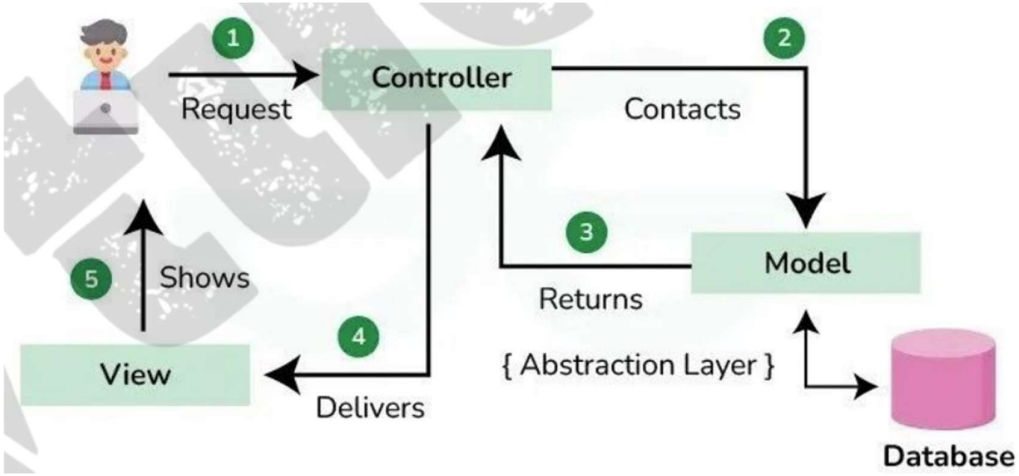
- Session Management: Web frameworks typically include mechanisms for managing user sessions, such as storing session data on the server or using client-side cookies to maintain session state.

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

MVC Design Pattern

The MVC pattern is a software architecture pattern that separates data presentation from the logic of handling user interactions(in other words, saves you stress:), it has been around as a concept for a while, and has invariably seen an exponential growth in use since its inception. It has also been described as one of the best ways to create client-server applications, all of the best frameworks for web are all built around the MVC concept

To break it down, here's a general overview of the MVC Concept;



Model: This handles your data representation, it serves as an interface to the data stored in the database itself, and also allows you to interact with your data without having to get perturbed with all the complexities of the underlying database.

Characteristics:

- It doesn't 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: As the name implies, it represents what you see while on your browser for a web application or In the UI for a desktop 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 .

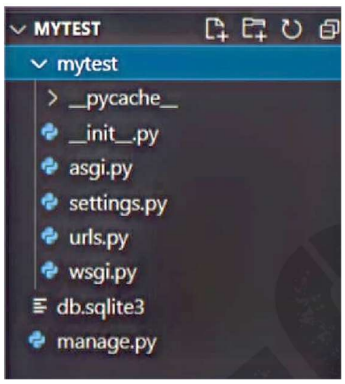
- Example: In a web application, the View component comprises HTML templates, CSS stylesheets, and client-side scripts (e.g., JavaScript) responsible for rendering dynamic content and handling user interactions.

Controller: provides the logic to either handle presentation flow in the view or update the model's data i.e it uses programmed logic to figure out what is pulled from the database through the model and passed to the view, also gets information from the user through the view and implements the given logic by either changing the view or updating the data via the model , To make it more simpler, see it as the engine room.

- 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

1.6 PROJECT STRUCTURE

django-admin startproject mytest
This will create a "myproject " folder with the following structure



- Project Directory: This is the main directory for your Django project. It contains everything related to your project.
- init_.py: this empty file tells Python that this folder is a Python package.
- 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 WSGl-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 ASGl-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: Th is 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.

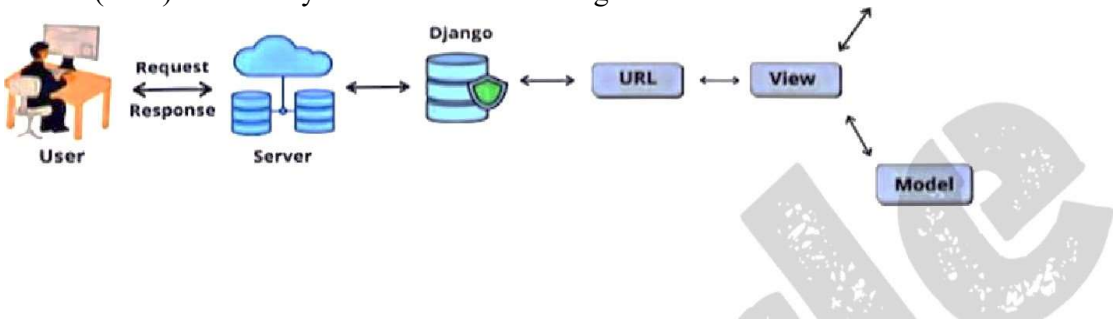
- Media Root: This is the directory where user-uploaded files are stored. It's configured in settings.py.

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.



Views in Django

Views

Django views are Python functions that takes http requests and returns http response, like HTML documents.

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("He/Io, World'")

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 HITP request.

return HttpResponse("HelIo, World l")

Line 3: 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

A web page that uses Django is full of views with different tasks and missions.Views are usually put in a file called views.py located on your app's folder.

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):

HttpResponse("Hello, World!")

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

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.

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.

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.

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 classbased view in Django to handle GET requests

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):
```

Example class-based view:

```
from django. views import View from
django.http import HttpResponse
class MyView(View):
def get(self, request):
    return HttpResponse("He/Io, 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( lhello-class/',
MyView.as_view(), name='he//o-
class'),
```

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.

Working of Django URL Confs and Loose Coupling

Now's a good time to highlight a key philosophy behind URLconfs and behind Django in general: the principle of loose coupling. Simply put, loose coupling is a software-development approach that values the importance of making pieces interchangeable. If two pieces of code are loosely coupled, then changes made to one of the pieces will have little or no effect on the other.

Django's URLconfs are a good example of this principle in practice. In a Django web application, the URL definitions and the view functions they call are loosely coupled – i.e., the decision of what the URL should be for a given function, and the implementation of the function itself, reside in two separate places.

For example, consider our current_datetime view. If we wanted to change the URL for the application – say, to move it from /time/ to /current-time/– we could make a quick change to the URLconf, without having to worry about the view itself. Similarly, if we wanted to change the view function – altering its logic somehow – we could do that without affecting the URL to which the function is bound. Furthermore, if we wanted to expose the current-date functionality at several URLs, we could easily take care of that by editing the URLconf, without having to touch the view code.

Loose Coupling in Django URI-confs:

- **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 IJRL 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.

Errors in Django

404 Errors

In our URLconf thus far, we've defined only a single URLpattern: the one that handles requests to the URL /time/. What happens when a different URL is requested?

To find out, try running the Django development server and hitting a page such as http://127.0.0.1:8000/hello/ or http://127.0.0.1:8000/does-not-exist/, or even http://127.0.0.1:8000/ (the site "root"). You should see a "Page not found" message (see Figure 3-2). (Pretty, isn't it? We Django people sure do like our pastel colors.) Django displays this message because you requested a URL that's not defined in your URLconf.

Figure 3-2. *Django's 404 page*

The utility of this page goes beyond the basic 404 error message; it also tells you precisely which URLconf Django used and every pattern in that URLconf. From that information, you should be able to tell why the requested URL threw a 404.

Naturally, this is sensitive information intended only for you, the Web developer. If this were a production site deployed live on the Internet, we wouldn't want to expose that information to the public. For that reason, this "Page not found" page is only displayed if your Django project is in *debug mode*. We'll explain how to deactivate debug mode later. For now, just know that every Django project is in debug mode when you first create it, and if the project is not in debug mode, a different response is given.

Wild Card patterns in URLS

Wild card patterns in URLs refer to a set of characters that can be used as placeholders for one or more other characters in a URL. The most common wild card character is the asterisk (*), which can represent any sequence of characters. Other wild card characters include the question mark (?) and the underscore (_), which can represent a single character or a hyphen. Wild card patterns can be used in URLs to make them more flexible and adaptable to different situations.

Wild card patterns in URLs are used to create flexible and adaptable URL structures by allowing placeholders for one or more characters. Here's how they are commonly used:

1**. Asterisk (*):** The asterisk is used to represent any sequence of characters in a URL. For example, if you have a URL pattern like `https://www.example.com/products/*/details`, the asterisk can match any sequence of characters in the placeholder position. So, URLs like `https://www.example.com/products/item1/details` and `https://www.example.com/products/123/details` would both match this pattern.

2. Question Mark (?): The question mark is used to represent a single character in a URL. For instance, a URL pattern like `https://www.example.com/search?q=term?` would match URLs like `https://www.example.com/search?q=term1` and `https://www.example.com/search?q=terma`.

3. Underscore (_) and Hyphen (-): These characters can also be used as wild cards to represent a single character in a URL. For instance, a URL pattern like `https://www.example.com/users/user_\_/profile` could match URLs like `https://www.example.com/users/user_a/profile` and `https://www.example.com/users/user_1/profile`.

By using wild card patterns in URLs, web developers can create more versatile and dynamic URL structures to accommodate various scenarios and user inputs.