WEB APPLICATION

What is a web application?

A web application is a type of software application that is accessed and used through a web browser over the internet. Unlike traditional desktop applications that are installed locally on a user's computer, web applications are hosted on remote servers, and users interact with them through their web browsers. Here are some key points about web applications and how they are useful:

- **Accessibility:** Web applications are accessible from anywhere with an internet connection. Users can access the application using different devices (e.g., computers, smartphones, tablets) and different operating systems, making them highly versatile and accessible.
- **Cross-Platform Compatibility:** Web applications are typically platform-independent. They can run on various web browsers and operating systems, reducing the need to develop and maintain different versions for different platforms.
- **No Installation Required:** Users don't need to install or update software on their local devices. This simplifies the user experience and reduces the burden of software installation and maintenance.
- **Automatic Updates:** Web applications are maintained on the server side. When updates or bug fixes are made, all users instantly benefit from the improvements without having to manually update their software.
- **Centralized Data and Storage:** Data is stored on the server, which makes it easy to centralize and manage. Users can access their data from any device, ensuring data consistency.
- **Collaboration:** Web applications are well-suited for collaborative work. Multiple users can access and work on the same data simultaneously, making them ideal for projects that require teamwork.
- **Scalability:** Web applications can scale easily to accommodate a growing number of users. This scalability is achieved by adding more server resources or using cloud-based solutions.
- **Security:** Many web applications implement robust security measures, including data encryption and user authentication, to protect user data and ensure a secure environment.
- **Cost-Efficiency:** Developing web applications can be cost-effective, especially for businesses, as it reduces the need to develop and maintain separate applications for different platforms.
- **Global Reach:** Web applications can reach a global audience. Users from around the world can access the application, expanding the potential user base.

- **Real-Time Updates:** Web applications can provide real-time updates and interactions. They can support features like chat, notifications, and live data streaming.
- **Analytics and User Insights:** Web applications can collect and analyze user data, providing valuable insights into user behaviour and preferences, which can be used for continuous improvement.
- **Customization:** Many web applications allow users to customize their experience, such as changing settings, preferences, and user profiles.

Common examples of web applications include email services like Gmail, social media platforms like Facebook, online shopping websites like Amazon, and cloud-based productivity tools like Google Docs.

In summary, web applications are valuable tools that provide users with easy and universal access to software and services, while offering businesses and developers an efficient way to deliver and maintain software across the internet. Their accessibility, cross-platform compatibility, and other advantages make them an integral part of our digital lives.

Django History

Django is a high-level Python web framework that was created to simplify the process of building web applications. Its history dates back to the early 2000s. Here's a basic overview of Django's history:

1. Development at Lawrence Journal-World (2003-2005):

- Django was initially developed by Adrian Holovaty and Simon Willison while working at the Lawrence Journal-World, a newspaper in Lawrence, Kansas.- The development of Django began in 2003 when the team wanted to build a content management system (CMS) for the newspaper's website. They decided to create a framework to make web development easier and more efficient.

2. Open Source Release (July 2005):

Django was released as an open-source project in July 2005. The release made it available to the wider developer community, and it quickly gained popularity.

3. Named After Jazz Guitarist Django Reinhardt:

The framework was named after the famous Belgian jazz guitarist Django Reinhardt. Holovaty and Willison were fans of his music and decided to name their project in his honour.

4. Growth in Popularity:

Django gained popularity rapidly due to its clean and pragmatic design, which emphasized the "batteries-included" philosophy, providing a wide range of built-in tools and features for web development.

5. Community and Ecosystem:

A strong and active community of developers and contributors formed around Django, leading to the development of numerous extensions, packages, and resources to enhance its functionality.

6. Releases and Evolution:

Django has had multiple major releases, each introducing new features and improvements. These releases have continued to refine and expand the framework's capabilities.

7. Adoption by Major Websites and Organizations:

Django has been adopted by numerous major websites and organizations, including Instagram, Pinterest, NASA, and The Washington Times, among many others. Its robustness and scalability have made it a choice for a wide range of web applications.

8. Internationalization and Localization:

- Django has been developed with a strong emphasis on internationalization and localization, making it suitable for building websites and applications in multiple languages and regions.

9. Community Conferences and Events:

The Django community organizes conferences and events worldwide, such as DjangoCon, to bring developers together, share knowledge, and discuss the latest developments in the Django ecosystem.

Django Web Application

A Django web application is a software application developed using the Django web framework, which is written in Python. Django is a high-level, open-source web framework that provides a structured and efficient way to build web applications. A Django web application typically follows the Model-View-Controller (MVC) architectural pattern, but in Django, it's referred to as the Model-View-Template (MVT) pattern. Here's a breakdown of the key components of a Django web application:

Models: In a Django application, models represent the data structure of your application. Models define the database schema, and you define them as Python classes. Django's Object-Relational Mapping (ORM) handles the translation of these models into database tables. This makes it easier to work with databases as you can interact with data using Python objects.

Views: Views in Django handle the logic of your application. They receive incoming web requests, process them, interact with the models if necessary, and return appropriate responses. Views can be implemented as Python functions or classes.

Templates: Templates are responsible for the presentation layer of your application. They define the HTML structure and the way data is presented to the user. Django's template engine allows you to create dynamic web pages by embedding Python code within HTML templates.

URLs: The URL dispatcher in Django maps URLs to views. You define URL patterns that specify which view function or class should be called when a particular URL is requested. This allows for clean and organized URL routing.

Settings: Django has a settings module where you configure various aspects of your application, such as database connections, installed apps, and middleware. It provides a central place to manage application settings.

Middleware: Middleware components in Django are used to process requests and responses globally across the application. You can use middleware to perform tasks like authentication, logging, and security checks.

Admin Interface: Django provides an admin interface that can be automatically generated based on your data models. This interface allows you to manage the content and data of your web application without having to build custom administrative tools.

Forms: Django includes a powerful form handling system that simplifies form creation and validation. It's often used to handle user input and data submission.

Django web applications are known for their speed of development, maintainability, and security. They follow the "batteries-included" philosophy, which means that Django provides a wide range of built-in tools and features to handle common web development tasks. This allows developers to focus on building the specific functionality of their applications rather than reinventing the wheel.

In summary, a Django web application is a web-based software application developed using the Django web framework, and it leverages Django's features and components to handle data, logic, and presentation in a structured and efficient manner.

Django web applications offer several advantages:

Rapid Development: Django follows the "batteries-included" philosophy, providing a wide range of built-in tools and features. This accelerates development, as developers don't need to reinvent the wheel for common web development tasks.

Security: Django has built-in security features to help protect web applications from common web vulnerabilities like cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection.

Scalability: Django is designed to scale, allowing applications to handle increased traffic and load through various deployment and optimization options.

Modularity: Django promotes a modular design, allowing developers to create reusable components and easily extend their applications with third-party packages and libraries.

Community and Documentation: Django has a vibrant and active community, which means access to resources, documentation, and a wealth of third-party packages and extensions.

Database Abstraction: The Object-Relational Mapping (ORM) system in Django abstracts database interactions, making it easier to work with databases without writing raw SQL queries.

Cross-Platform Compatibility: Django applications are platform-independent and can run on various web servers and operating systems.

Internationalization and Localization: Django has strong support for building multilingual and region-specific web applications.

Admin Interface: The admin interface provides an out-of-the-box solution for managing the content and data of your application, which is a time-saver for developers.

Installing Django

Introduction: This guide will walk you through the step-by-step process of installing Django on your Windows computer, allowing you to start building web applications using the Django web framework.

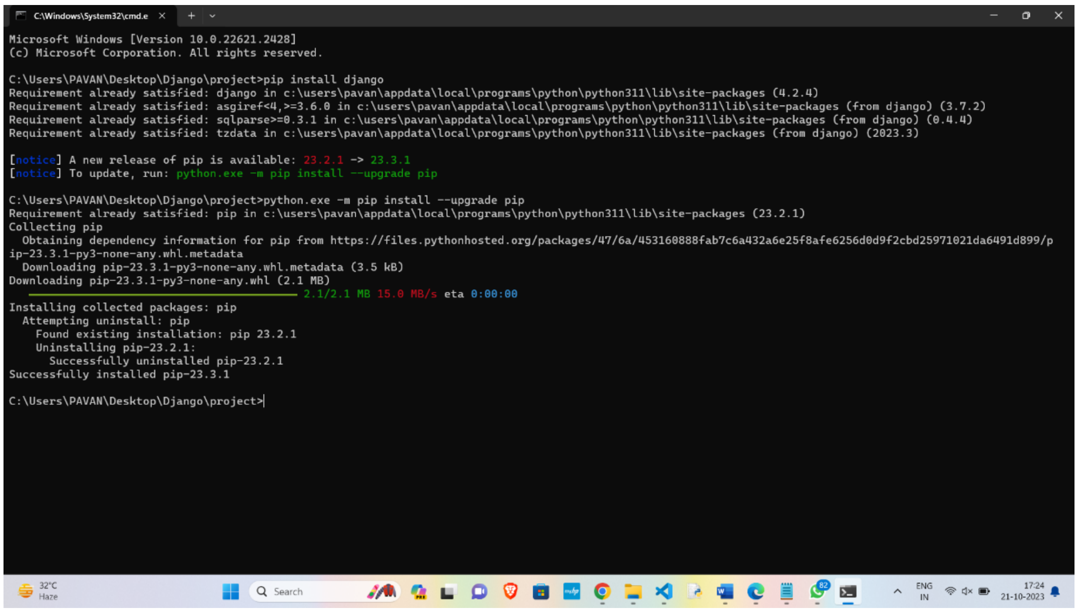
Prerequisites: Before you begin, ensure you have Python installed on your windows computer. If Python is not installed, download and install it from Python's official website.

Django requires Python 3.6 or later, so make sure you have an appropriate version.

Installing Django: Open the Windows Command Prompt (you can search for "cmd" in the Start menu).

To install Django, use Python's package manager pip. Run the following command:

```
bash
pip install Django
```



```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\PAVAN\Desktop\Django\project>pip install django
Requirement already satisfied: django in c:\users\pavan\appdata\local\programs\python\python311\lib\site-packages (4.2.4)
Requirement already satisfied: asgiref<4,>=3.6.0 in c:\users\pavan\appdata\local\programs\python\python311\lib\site-packages (from django) (3.7.2)
Requirement already satisfied: sqlparse>=0.3.1 in c:\users\pavan\appdata\local\programs\python\python311\lib\site-packages (from django) (0.4.4)
Requirement already satisfied: tzdata in c:\users\pavan\appdata\local\programs\python\python311\lib\site-packages (from django) (2023.3)

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\PAVAN\Desktop\Django\project>python.exe -m pip install --upgrade pip
Requirement already satisfied: pip in c:\users\pavan\appdata\local\programs\python\python311\lib\site-packages (23.2.1)
Collecting pip
  Obtaining dependency information for pip from https://files.pythonhosted.org/packages/47/6a/453160888fab7c6a432a6e25f8afe6256d0d9f2cbd25971021da6491d899/p
ip-23.3.1-py3-none-any.whl.metadata
  Downloading pip-23.3.1-py3-none-any.whl.metadata (3.5 kB)
Downloading pip-23.3.1-py3-none-any.whl (2.1 MB)
   ---------------------------------------- 2.1/2.1 MB 15.0 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 23.2.1
    Uninstalling pip-23.2.1:
      Successfully uninstalled pip-23.2.1
Successfully installed pip-23.3.1

C:\Users\PAVAN\Desktop\Django\project>
```

This will download and install Django on your computer.

Verify Installation: To ensure that Django is installed correctly, you can check the installed version. Run the following command:

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

This command should display the version of Django you installed.

Optional - Create a Virtual Environment:

It's a good practice to work within a virtual environment to isolate your Django project from other Python projects.

To create a virtual environment, use the following commands:

bash

```
mkdir mydjangoenv
cd mydjangoenv

venv\Scripts\activate
```

This creates a virtual environment and activates it.

Optional - Creating a Django Project:

To create a new Django project, navigate to the directory where you want to create your project and run:

bash

```
django-admin startproject projectname
```

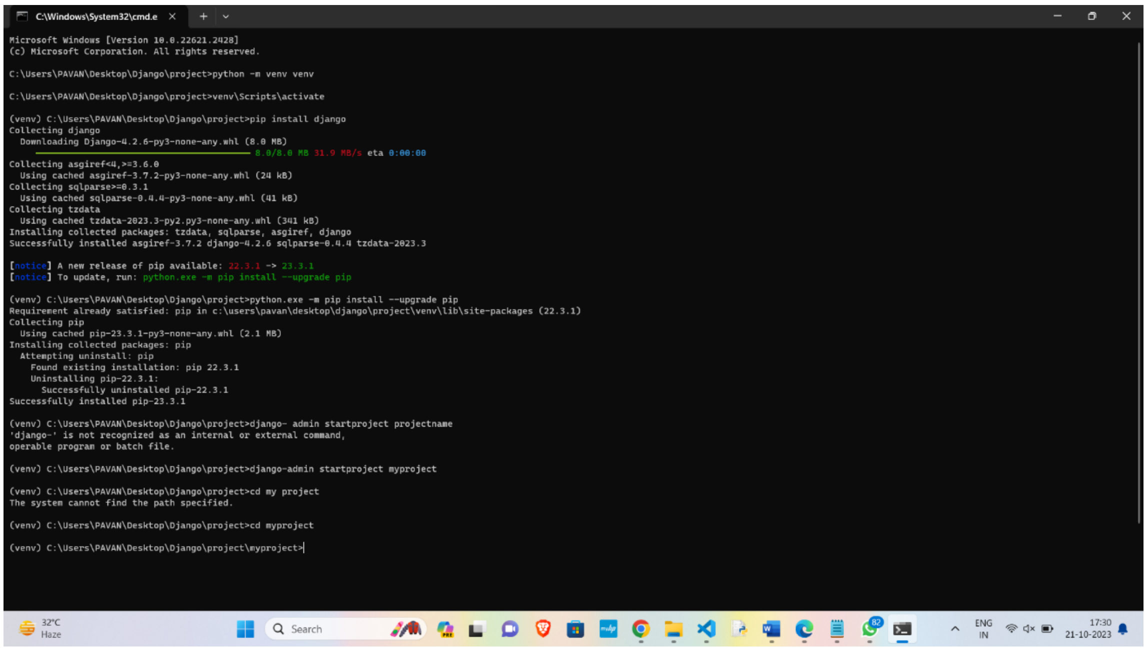
This will create a new Django project with the given name.

Optional - Deactivate the Virtual Environment:

If you created a virtual environment in Chapter 4, it's a good practice to deactivate it when you're done working on your Django project. To do so, run:

bash

deactivate



```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\PAVAN\Desktop\Django\project>python -m venv venv

C:\Users\PAVAN\Desktop\Django\project>venv\Scripts\activate

(venv) C:\Users\PAVAN\Desktop\Django\project>pip install django
Collecting django
  Downloading Django-4.2.6-py3-none-any.whl (8.0 MB)
     ---------------------------------------- 8.0/8.0 MB 31.9 MB/s eta 0:00:00
Collecting asgiref<4,>=3.6.0
  Using cached asgiref-3.7.2-py3-none-any.whl (24 kB)
Collecting sqlparse>=0.3.1
  Using cached sqlparse-0.4.4-py3-none-any.whl (41 kB)
Collecting tzdata
  Using cached tzdata-2023.3-py2.py3-none-any.whl (341 kB)
Installing collected packages: tzdata, sqlparse, asgiref, django
Successfully installed asgiref-3.7.2 django-4.2.6 sqlparse-0.4.4 tzdata-2023.3

[notice] A new release of pip available: 22.3.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip

(venv) C:\Users\PAVAN\Desktop\Django\project>python.exe -m pip install --upgrade pip
Requirement already satisfied: pip in c:\users\pavan\desktop\django\project\venv\lib\site-packages (22.3.1)
Collecting pip
  Using cached pip-23.3.1-py3-none-any.whl (2.1 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 22.3.1
    Uninstalling pip-22.3.1:
      Successfully uninstalled pip-22.3.1
Successfully installed pip-23.3.1

(venv) C:\Users\PAVAN\Desktop\Django\project>django- admin startproject projectname
'django-' is not recognized as an internal or external command,
operable program or batch file.

(venv) C:\Users\PAVAN\Desktop\Django\project>django-admin startproject myproject

(venv) C:\Users\PAVAN\Desktop\Django\project>cd my project
The system cannot find the path specified.

(venv) C:\Users\PAVAN\Desktop\Django\project>cd myproject

(venv) C:\Users\PAVAN\Desktop\Django\project\myproject>
```

Conclusion:

You have successfully installed Django on your Windows computer. You are now ready to start developing web applications using this powerful web framework.

End of Guide

Feel free to format and structure this information as needed for your notes or book-style document. These steps should help you install Django on your Windows computer and begin your journey into web development using the framework.

Title: Creating a Django Project from the Beginning

Introduction:

In this guide, we will start from the very beginning and create a Django project using a file manager. This approach is suitable for those who prefer a visual file management process.

Prerequisites:

- Before we begin, make sure you have Python and Django installed on your system. If not, refer to the guide on "Installing Django" to set up your environment.

Folder Creation:

- Open your file manager (e.g., Windows Explorer, macOS Finder).

- Choose or create a directory where you want to create your Django project. You can create a new folder using the file manager's options.

Command Prompt (Optional):

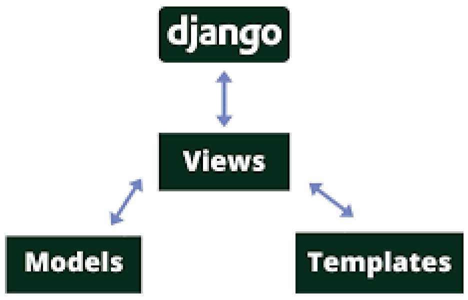
- While we'll primarily use the file manager, you can also open a command prompt or terminal and navigate to the project directory. This is optional but useful for running Django management commands.

Create the Project Directory:

- Inside your chosen directory, create a new folder. This will be the root directory of your Django project. Give it a name, e.g., `myproject`.

MVT Architecture

It stands for **Model View Template** Architecture.



Architecture is the process of designing, creating and implementing an internet-based computer program. Often, these programs are websites that contain useful information for a user, and web developers may design these programs for a particular purpose, company or brand.

Views:

In Django, the views.py file is a key component of the Model-View-Controller (MVC) architectural pattern used to build web applications. Views are responsible for handling HTTP requests, processing data, and returning appropriate HTTP responses. Here's a simple explanation of what views.py is and its main concepts:

1. **Request Handling:** Views in Django are Python functions or classes that handle incoming HTTP requests. They contain the application logic for processing requests from clients (web browsers, mobile apps, etc.).
2. **Data Processing:** In views, you can perform various tasks such as retrieving data from a database, applying business logic, rendering templates, and more. This is where you determine what data to display to the user.
3. **HTTP Responses:** Views are also responsible for creating and returning HTTP responses. This includes rendering HTML templates, returning JSON data for AJAX requests, or performing redirects, among other responses.
4. **Function-Based and Class-Based Views:** In Django, you can define views as simple Python functions or as classes. Function-based views are straightforward functions, while class-based views offer a more organized way to handle views, providing methods for different HTTP request methods (e.g., get, post, put, etc.).

Here's a simple example of a function-based view in views.py that renders a basic "Hello, World!" response:

```
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse("Hello, World!")
```

In the above example:

The hello_world function is a view that takes an HTTP request as an argument (usually named request).

It returns an HttpResponse object containing the text "Hello, World!" as the response content. This content will be sent to the client's browser when the view is accessed.

To use a view, you typically map it to a URL pattern in Django's URL configuration (urls.py). When a user accesses a specific URL, the associated view function or class is executed, and the response is generated based on the logic defined in the view.

View functions are the heart of the application's logic. They decide what to show, process user input, and interact with the database or other data sources. Django's URL routing system directs incoming requests to the appropriate views, allowing you to build dynamic and interactive web applications.

URLS:

URL is a path through which a specific web-based application and one particular page in that web application can be reached.

In Django, URLs, also known as URL patterns, are a fundamental component of the web framework that determine how web requests are routed to specific views within your application. Here's a simple explanation of URLs in Django:

1. **URL Routing:** URLs in Django are used to map specific web addresses (URLs) to Python functions or classes known as views. These views are responsible for handling requests and generating responses.
2. **URL Patterns:** URL patterns define the structure of the URLs in your application. They are defined in the urls.py file of your Django app. Each URL pattern is associated with a particular view.
3. **View Dispatch:** When a user visits a URL in your application, Django's URL router matches the URL to a defined pattern and calls the associated view. The view then processes the request and returns an HTTP response.
4. **Regular Expressions:** URL patterns are often defined using regular expressions (regex) to provide flexibility in matching URLs. This allows for dynamic and parameterized URLs that can capture and pass data to views.

Here's a simple example of URL patterns in Django:

In your urls.py:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
    path('about/', views.about, name='about'),
    path('blog/<int:post_id>/', views.blog_detail, name='blog_detail'),
]
```

In the above example:

path('') maps the root URL to the home view.

path('about/') maps the URL "/about/" to the about view.

path('blog/<int:post_id>/') maps URLs like "/blog/1/" or "/blog/42/" to the blog_detail view, capturing the post_id as an integer parameter.

In your views.py:

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Welcome to the home page!")
def about(request):
    return HttpResponse("Learn more about us on the about page!")
def blog_detail(request, post_id):
    return HttpResponse(f"You are reading blog post #{post_id}")
```

In this example, each view function returns a simple HTTP response. When a user visits a URL like "/about/", the about view is called, and when they visit a URL like "/blog/1/", the ‘blog_detail’ view is called with ‘post_id’ as a parameter.

Django's URL routing system helps organize and structure your web application, making it easy to map specific URLs to the appropriate views for processing and responding to user requests.

Django URL Error codes:

400	Bad request
403	Permission denied
404	Page not found
500	Server Error

Models:

In simple words a Django Model is Nothing But a table in the database

The data in Django created in objects called as models and are actually tables in Database.

By Default, Django Provides SQLite as Database.

In Django, the models.py file is a crucial component of the Model-View-Controller (MVC) architecture used to build web applications. It defines the structure of your application's database tables and how data is stored and retrieved. Here's a simple explanation of what models.py is and its key concepts:

1. **Database Tables:** In a Django application, data is typically stored in a relational database. Each table in the database corresponds to a model defined in models.py. A model is a Python class that represents a specific type of data, such as a user, a product, a blog post, etc.
2. **Fields:** Inside a model class, you define fields to represent the attributes of the data you want to store. These fields specify what kind of data can be stored in the database table. Django provides various field types like CharField, IntegerField, DateField, ForeignKey, and more, which correspond to different data types like strings, numbers, dates, and relationships between tables.
3. **Model Methods:** You can define methods within your model class to perform various operations related to that model. These methods can be used to manipulate data before it's saved to the database, perform calculations, or execute other custom logic.
4. **Data Validation:** Models can also include data validation by setting constraints on the fields. For example, you can define a field with max_length or unique properties to restrict the length of a string or ensure that a field's value is unique across all records.

Here's a simple example of a models.py file for a blog application:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    pub_date = models.DateTimeField('date published')
    def __str__(self):
        return self.title

    def get_absolute_url(self):
        # Define a method to get the URL of a specific post
        return reverse('post_detail', args=[str(self.id)])
```

In the above example:

Post is a model representing blog posts.

title, content, and pub_date are fields that store the title, content, and publication date of a blog post, respectively.

The __str__ method is used to provide a human-readable representation of the model when it's displayed in the Django admin interface or other contexts.

The get_absolute_url method can be used to generate the URL for a specific blog post.

Once you've defined your models in models.py, you can use Django's Object-Relational Mapping (ORM) to interact with the database and perform operations like creating, retrieving, updating, and deleting records. Django takes care of translating your model definitions into SQL queries for the database, making it a powerful tool for working with databases in web applications.

Templates

In Django, templates are a fundamental part of the Model-View-Controller (MVC) architecture used to build web applications. They serve as a way to separate the presentation (how content is displayed) from the application logic (how data is processed and handled). Here's a simple explanation of what templates are in Django:

1. **HTML with Special Tags:** A Django template is essentially an HTML file that contains special template tags and filters. These tags and filters are used to insert dynamic data, logic, and control structures into the HTML.
2. **Dynamic Content:** Templates allow you to display data from your application's models and views dynamically. For example, you can use template tags to insert the title of a blog post, the username of a logged-in user, or a list of products from a database.
3. **Reuse and Extensibility:** Templates promote reusability. You can create a base template that defines the common structure of your site (e.g., header and footer) and then extend it with more specific templates for

individual pages or sections. This makes it easy to maintain a consistent look and feel across your web application.

4. **Control Structures:** Templates also support control structures like if statements and loops, allowing you to conditionally display content or iterate over lists of data. For instance, you can use an if statement to show different content to logged-in and anonymous users.

Here's a simple example of a Django template:

```
<!DOCTYPE html>
<html>
<head>
   <title>{{ page_title }}</title>
</head>
<body>
   <header>
      <h1>Welcome to My Website</h1>
   </header>

   <main>
      <h2>{{ post.title }}</h2>
      <p>{{ post.content|linebreaks }}</p>
   </main>

   <footer>
      <p>&copy; 2023 My Website</p>
   </footer>
</body>
</html>
```

In this example:

{{ page_title }} and {{ post.title }} are template tags that will be replaced with actual data when the template is rendered.

{{ post.content|linebreaks }} demonstrates the use of a filter (linebreaks) to format the content.

HTML structure, such as <head>, <header>, and `<footer>, is preserved, and only the dynamic content is inserted.

To use a template, you pass context data from your Django views to the template. The template engine then replaces the template tags and filters with the actual data, and the resulting HTML is sent to the client's browser. This separation of concerns makes it easier to maintain and scale web applications, as it allows developers and designers to work on different aspects of a project independently.

Difference between MVC and MVT

Django, a web framework, follows the MVT (Model-View-Template) architectural pattern, which is conceptually similar to the more commonly known MVC (Model-View-Controller) pattern. However, there are some differences in terminology and emphasis. Here's a breakdown of the main distinctions between MVC and MVT in the context of Django:

MVC (Model-View-Controller):

Model (M): The Model represents the data and business logic of an application. It defines how data is structured and how it should be manipulated. In traditional MVC, the Model directly communicates with the Controller.

View (V): The View is responsible for displaying the data to the user. It defines the user interface and presentation. In traditional MVC, the View communicates with the Model to retrieve data for presentation.

Controller (C): The Controller acts as an intermediary between the Model and the View. It receives user input, processes it, and instructs the Model to update data accordingly. In traditional MVC, the Controller plays a central role in managing the flow of data and user interactions.

MVT (Model-View-Template) in Django:

Model (M): The Model in Django, like in MVC, defines the data structure and database schema. It handles data storage and retrieval and contains business logic. The key difference is that in Django, the Model interacts with the database directly.

View (V): In Django's MVT, the View is responsible for processing HTTP requests and returning HTTP responses. This includes handling user interactions, data processing, and rendering templates. The View in Django corresponds to both the Controller and View in traditional MVC.

Template (T): Django introduces the Template component, which is responsible for defining how data is presented in the user interface. Templates are HTML files with embedded placeholders for dynamic data. This corresponds to the presentation layer and is similar to the View in traditional MVC.

Key Differences:

In traditional MVC, the Controller is responsible for managing the flow of data and user interactions. In Django's MVT, the View plays this role, while the Template focuses exclusively on defining the presentation.

Django's approach separates the responsibility of generating dynamic HTML (Template) from handling user input and processing data (View).

Django emphasizes reusability and modularity by keeping the View and Template separate, making it easier to work with designers for the user interface.

In summary, both MVC and MVT are architectural patterns that aim to separate the concerns of an application, but Django's MVT introduces its own terminology and emphasizes a clear separation between data processing (View) and presentation (Template). The Model remains responsible for data management in both patterns.

Protocols

In Django, a "protocol" usually refers to a set of rules and conventions followed by the framework and web applications built with Django. These protocols help facilitate communication between different parts of a Django application, ensuring that they work together smoothly. Some common examples of protocols and conventions in Django are follows:

URL Routing Protocol: Django uses a URL routing protocol to map specific URLs to views and functions that handle those URLs. This protocol defines how URLs are structured and how they should be matched to specific view functions.

HTTP Protocol: Django applications are built on top of the HTTP (Hypertext Transfer Protocol). The HTTP protocol defines how web clients (browsers) and servers communicate, including the methods for requesting and responding to data.

Model-View-Controller (MVC) Protocol: Django follows the MVC architectural pattern, where models, views, and controllers (handled by Django's views) interact to process and display data. This protocol defines how data flows between these components.

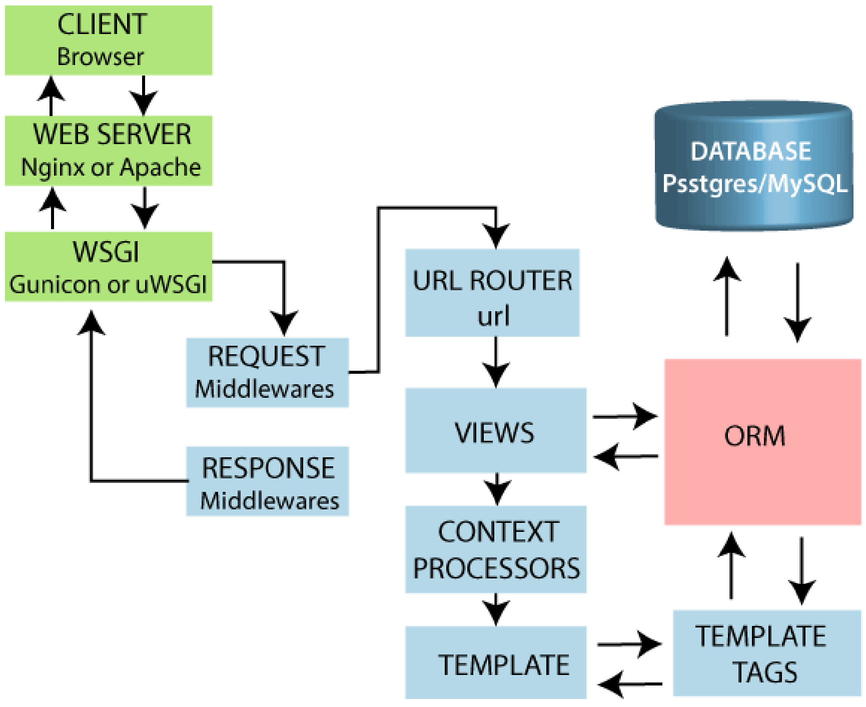
Template Protocol: Django templates follow a specific protocol for rendering dynamic HTML. This includes using template tags, filters, and context data to generate HTML pages.

Database Access Protocol: Django provides an Object-Relational Mapping (ORM) layer that abstracts database interactions. This protocol defines how you interact with the database through Python code, rather than writing raw SQL queries.

Form Handling Protocol: Django provides a protocol for creating and processing web forms, which includes form classes, form validation, and form submission handling.

Middleware Protocol: Middleware components in Django follow a protocol for processing requests and responses globally across the application. Developers can create custom middleware to perform tasks such as authentication and logging.

These protocols and conventions help maintain consistency, structure, and best practices in Django applications. They make it easier for different developers to work on the same codebase and help developers understand how to interact with various parts of the framework.



The following are the some of the key protocols and standards used in Django:

HTTP (Hypertext Transfer Protocol): It's like the language that web browsers and servers use to talk to each other. When you visit a website, your browser sends an HTTP request to the server, which responds with the web page you see.

HTTPS (Hypertext Transfer Protocol Secure): This is like a secure version of HTTP. It's used to make sure the data exchanged between your browser and the server is encrypted and can't be easily intercepted by others.

URLs (Uniform Resource Locators): These are like web addresses that help you find and access specific web pages. In Django, we use URLs to map web addresses to the code that handles them.

HTML (Hypertext Markup Language): It's like the building blocks of web pages. HTML defines the structure and content of a web page, such as headings, paragraphs, and links.

CSS (Cascading Style Sheets): Think of CSS as the clothing for web pages. It's used to make web pages look nice by specifying colors, fonts, and layouts.

JavaScript: JavaScript is like the magic wand of the web. It adds interactivity to web pages, making them do things like showing pop-up messages or updating information without needing to reload the whole page.

AJAX (Asynchronous JavaScript and XML): AJAX is like a way to have secret conversations with the server while you're still using a web page. It's used to update parts of a page without making it flicker or reload.

Cookies and Sessions: These are like memory aids for websites. Cookies remember things about you, like your login status, and sessions keep track of your actions while you're using a site.

REST (Representational State Transfer): REST is like a set of rules for how web services should work. In Django, you can use REST to create web services that provide and receive data in a predictable and organized way.

WebSockets: WebSockets are like super-fast phone lines for real-time conversations between your browser and the server. They're used for things like live chats and online games.

Django handles all these technical details for you, making it easier to build web applications without having to be an expert in all these protocols. It provides a framework for creating web applications with less hassle and more focus on the actual features you want to build.

CREATING A PROJECT IN DJANGO:

Once you have come up with a suitable name for your Django project, like mine: projectName, navigate to where in the file system you want to store the code (in the virtual environment), I will navigate to the myworld folder, and run this command in the command prompt:

```
django-admin startproject projectName
```

Django creates a projectName folder on my computer, with this content:

```
projectName
    manage.py
    projectName/
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
```

The project configuration typically includes the following files:

manage.py:

A command-line utility script used for various administrative tasks, such as running the development server, creating database tables, and running management commands.

settings.py:

The main configuration file for your Django project. It contains settings like database configuration, middleware, authentication, installed apps, and more.

Custom settings can be added here as well.

urls.py:

Defines the URL patterns and routing for your application. It maps URL patterns to view functions or class-based views.

wsgi.py:

Configuration for the Web Server Gateway Interface (WSGI) server, which is used to serve your Django application through a web server like Apache or Nginx.

asgi.py:

Configuration for the Asynchronous Server Gateway Interface (ASGI) server, which is used for handling asynchronous tasks and real-time applications.

If we want to run the server you should use the below command on terminal

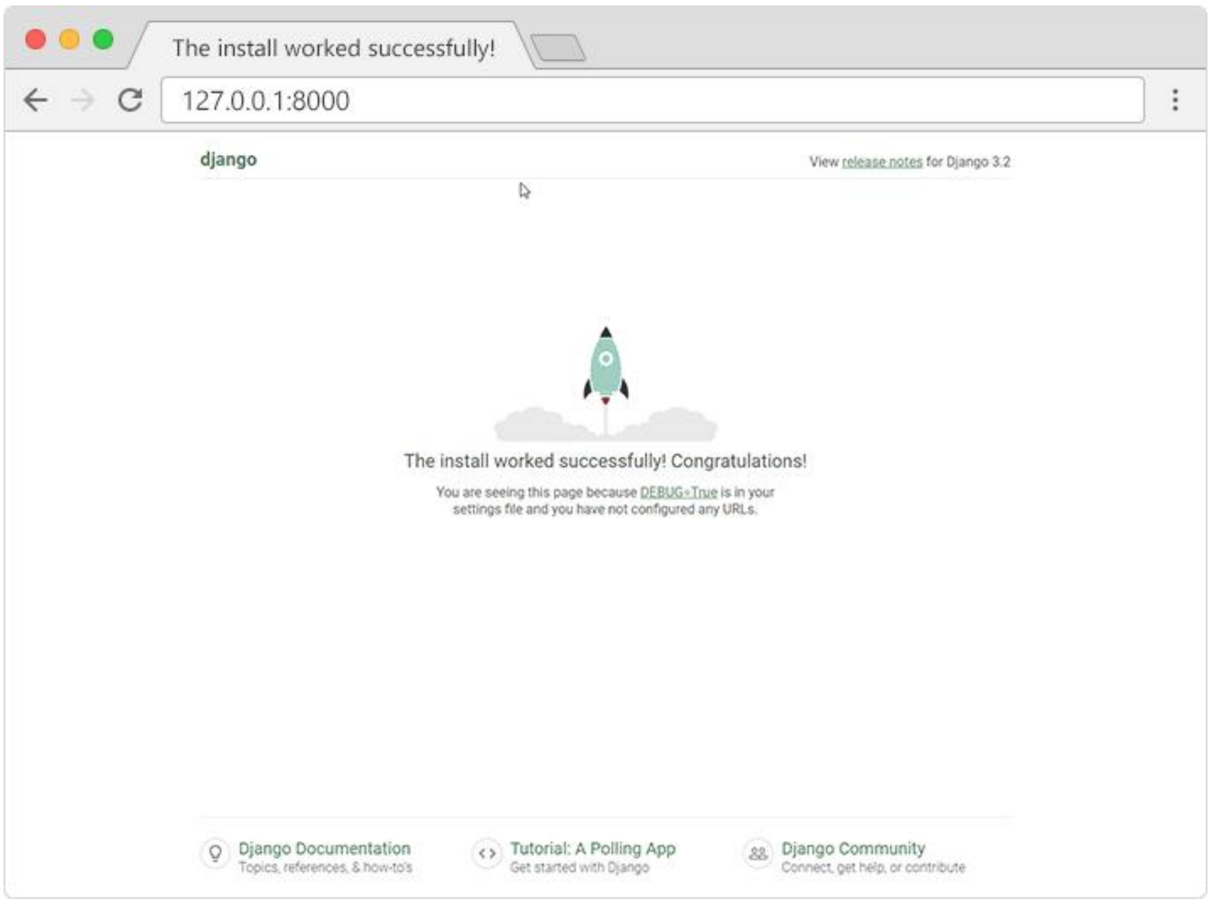
Navigate to Your Project Directory:

Use the cd command to move into the project directory:

>>cd projectname

python manage.py runserver

by clicking http://localhost:8000/ in your browser shown below


The install worked successfully!
127.0.0.1:8000
django
View release notes for Django 3.2
Django Documentation
Topics, references, & how-to's
Tutorial: A Polling App
Get started with Django
Django Community
Connect, get help, or contribute

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Creating an App in Django

Navigate to Your Project Directory:

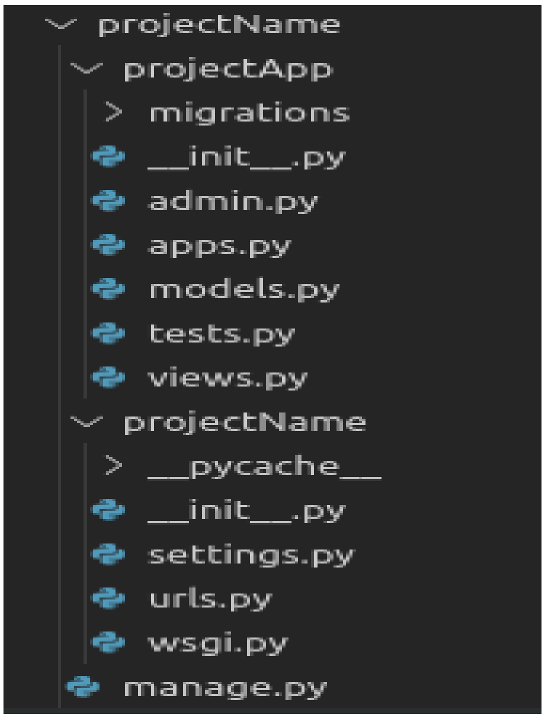
Use the cd command to move into the project directory:

>>cd projectname

To create a basic app in your Django project you need to go to the directory containing manage.py and from there enter the command :

>>python manage.py startapp projectApp

Now you can see your directory structure as under :



```
projectName
  projectApp
    migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  projectName
    __pycache__
    __init__.py
    settings.py
    urls.py
    wsgi.py
  manage.py
```

To consider the app in your project you need to specify your project name in INSTALLED_APPS list as follows in settings.py:

```
#settings.py
INSTALLED_APPS = [
	'django.contrib.admin',
	'django.contrib.auth',
	'django.contrib.contenttypes',
	'django.contrib.sessions',
	'django.contrib.messages',
	'django.contrib.staticfiles',
	'projectApp' # configure your app name here
]
```

So, we have finally created an app but to render the app using URLs we need to include the app in our main project so that URLs redirected to that app can be rendered. Let us explore it.

Move to projectName -> urls.py and add below code in the header

```
>>from djago.urls import include
```

Now in the list of URL patterns, you need to specify the app name for including your app URLs. Here is the code for it

```
from django.contrib import admin
from django.urls import path, include
#urls.py in Project Name
urlpatterns = [
	path('admin/', admin.site.urls),
	# Enter the app name in following
	# syntax for this to work
	path('firstproject', include("projectApp.urls")),
]
```

Now You can use the default MVT model to create URLs, models, views, etc. in your app and they will be automatically included in your main project.

The main feature of Django Apps is independence, every app functions as an independent unit in supporting the main project.

Now the urls.py in the project file will not access the app's url.

To run your Django Web application properly the following actions must be taken:-

1. Create a file in the apps directory called urls.py

2. Include the following code:

```
# urls.py projectApp
from django.urls import path
#now import the views.py file into this code
from . import views
```

```
urlpatterns=[
path('',views.index)
]
```

The above code will call or invoke the function which is defined in the views.py file so that it can be seen properly in the Web browser. Here it is assumed that views.py contains the following code :-

```
#views.py
from django.http import HttpResponse

def index(request):                          #by taking request as default
        return HttpResponse("Hello Guys")
```

Run the Development Server:

Start the Django development server with the following command:

```
>>python manage.py runserver
```

Go to http://localhost:8000/firstproject/ in your browser, and you should see the text “Hello Guys”. You’re at the firstprojectindex.”, which you defined in the index view.

And then you can run the server by clicking (127.0.0.1:8000) and you will get the desired output

Output:

“Hello Guys”. # it shows in the respective browser in your system

STATIC FILES:

Static files in web development refer to files that don't change or are not generated dynamically by the web application. These files are typically served directly to the client's web browser and include various types of assets that enhance the user experience and appearance of a web page.

The most common types of static files include:

CSS (Cascading Style Sheets): CSS files are used to define the styling and layout of a web page. They control the fonts, colors, spacing, and overall presentation of the content.

JavaScript (JS) Files: JavaScript files contain client-side code that adds interactivity and dynamic behavior to a web page. They are responsible for features like form validation, animations, and handling user interactions.

Images: Static files often include image files (e.g., JPEG, PNG, GIF) used for graphics, icons, logos, and other visual elements on a website.

Fonts: Font files (e.g., TTF, WOFF) are used to define custom typography and text styling on a web page.

Icons: Icon files (e.g., SVG, PNG) are commonly used for representing various actions, such as navigation icons or social media icons.

HTML Templates: While HTML itself can be considered dynamic, the base HTML templates that provide the structure of a web page are often static files. These templates are used as a foundation for dynamic content, which is injected into the templates on the server side.

Static files are typically served directly by web servers or content delivery networks (CDNs) to improve loading times and reduce server load. They are separated from dynamic content, which is generated and personalized for each user's request. This separation is essential for performance optimization and scalability.

In the context of Django and other web frameworks, developers need to manage and serve these static files to enhance the appearance and functionality of their web applications. Django provides tools and conventions for handling static files, as described in the previous answer, to simplify the process of serving CSS, JavaScript, images, and other assets.

HOW TO CONFIGURE THE STATIC FILES:

You need to create folder in our project :

```
mkdir static
mkdir static/css
mkdir static/js
mkdir static/img
```

First, open settings.py file and go straight to the STATIC_URL setting, which should be like this.

#Import os module in settingys.py

```
>>STATIC_URL = '/static/'
```

Next, we need to declare the STATICFILES_DIRS

```
STATIC_URL = '/static/'#Location of static files
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static'), ]
```

This tells Django the location of static files in our project. The common practice is to have all the static files in a top-level static directory.

STATICFILES_DIRS being a list indicates that having multiple static directories is possible.

Next, we have to set the STATIC_ROOT

```
STATIC_ROOT  = os.path.join(BASE_DIR, 'staticfiles')
```

Loading static files in Templates:

Open 'name'.html file and at the very top add {% load static % }

```
{% load static %}
<!DOCTYPE html><html>
<head>
  <title>Django Central</title>
```

This tells Django to load static template tags inside the template so you use the {% static %} template filer to include different static assets.

For example, in order to include the newly created base.css file in my template, I have to add this inside the <head> of the HTML document

```
<link rel="stylesheet" href="{% static 'css/base.css' %}">
```

Save and reload the page, you should see the change.

Similarly, you can load different scripts and images in templates.

```
<img src="{% static 'img/path_to_the_img' %}">
<script src="{% static 'img/path_to_the_script' %}"></script>
```

MEDIA FILES CONFIGURATIONS:

Open settings.py file of your project and add the following configuration.

SERVING MEDIA FILES IN CONFIGURATIONS:

In order to make the development server serve the media files open the url.py of the project and make the below changes.

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    ...]
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

DTL(Django Template Language):

In Django, "Django Template Language" (DTL) is the template engine used to create dynamic web pages and render content within HTML templates. DTL is a lightweight template language specifically designed for Django and is used to insert dynamic data, control structures, and filters into HTML templates.

Template Tags: DTL uses template tags enclosed in curly braces and percentage signs, such as {% tag_name %}. These tags provide control structures, loops, and other logic within templates. For example, {% for item in items %}...{% endfor %} is used for iterating over a list of items.

Variables: Variables can be inserted into templates using double curly braces, like {{ variable_name }}. These placeholders are replaced with actual data when the template is rendered.

Template Inheritance: Django templates support inheritance, allowing you to create a base template with common elements and extend it in child templates. This is useful for creating consistent layouts.

Static and Media Files: DTL provides template tags like {% static %} and {% media %} to reference static and media files respectively. These tags ensure that the correct URLs are generated for these files.

Conditions: DTL supports conditional statements, such as {% if condition %}...{% else %}...{% endif %}, allowing you to conditionally render content based on specific criteria.

Example:

```
{% if <condition1> %}
   {{<variable1>}}
{% elif <condition2> %}
   {{<variable2>}}
{% else %}
   {{<variable3>}}
{% endif %}
```

For Loops: You can use {% for item in items %}...{% empty %}...{% endfor %} to loop through lists or querysets. The empty block is executed if the list is empty.

Example:

```
#for loop

{% for i in data %}

        <div class="row">

                {{ i }}

        </div>

{% endfor %}
```

Blocks and Extends: DTL allows you to define blocks in your templates using {% block block_name %}...{% endblock %} and then extend a base template using {% extends "base_template.html" %}.

#First.html

```
<body>


   {% block content %}


   {% endblock %}


  </body>
</html>
```

#second.html

```
{% extends 'main/header.html' %}


<!--Block content goes below-->



  {% block content %}


  <h1>Hello, world!</h1>
  ...


  {% endblock %}
```

Comments: Comments can be added to your templates using {# This is a comment #}. These comments are not rendered in the final HTML.

URLs: You can use the {% url 'view_name' arg1 arg2 %} tag to generate URLs based on the view's name and arguments.

Example:

```
{% url 'View_Name' variable1 variable2 ... %}

<H2> The Webpage2 </H2>

<a href = "{% url 'webpage1' %}"> Go back </a>
```

DTL is a powerful and flexible template language that allows you to create dynamic, data-driven web pages in Django. It's designed to keep the logic in the views and models and the presentation in the templates, promoting clean and maintainable code. When taking notes for Django development, be sure to include these key concepts and syntax elements related to DTL to help you create effective and responsive web interfaces.

ORM(Object Relation Mapper):

ORM is a technique that allows you to manipulate data in a relational database using object-oriented programming. Django ORM allows you to use the same Python API to interact with various relational databases including PostgreSQL, MySQL, Oracle, and SQLite.

Connecting to the MySQL from Django:

First, configure the database connection in the settings.py of the ‘project name’ project:

#settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql ',
        'NAME': 'hr',       #Your database name
        'USER': 'root',      #your user name
        'PASSWORD': 'PASSWORD',
        'HOST': 'localhost',
        'PORT': '',    # your mysql port Number
    }
}
```

Creating a Model:

In app we have models.py here we need to configure the details:

Instead of writing the sql queries we create classes in that models.py

Here orm will convert the classes & objects into sql queries

```
from django.db import models

class Dreamreal(models.Model):

   website = models.CharField(max_length = 50)
   mail = models.CharField(max_length = 50)
   name = models.CharField(max_length = 50)
   phonenumber = models.IntegerField()

   class Meta:
      db_table = "dreamreal"
```

if we changes in models it is mandatory to do makemigrations and migrate

classes and objects converts sql queries

makemigrations:

Think of it as the "plan maker." When you make changes to your website's database (like adding a new feature or changing existing ones), this command helps create a detailed plan for what needs to change in the database to match your website's new design.

```
>>python manage.py makemigrations
```

migrate:

Think of it as the "action taker." Once you have a plan created with makemigrations, this command actually makes the changes to the database. It's like following a recipe to cook a meal – it takes the ingredients (the plan) and cooks the dish (updates the database).

All queries are migrate into sql tables

```
>>python manage.py migrate
```

In simple terms, makemigrations plans the changes, and migrate carries out those plans to update the database to match your website's requirements.

```
$python manage.py shell

>>> from myapp.models import Dreamreal
>>> dr1 = Dreamreal()
>>> dr1.website = 'company1.com'
>>> dr1.name = 'company1'
>>> dr1.mail = 'contact@company1'
>>> dr1.phonenumber = '12345'
>>> dr1.save()
>>> dr2 = Dreamreal()
>>> dr1.website = 'company2.com'
>>> dr2.website = 'company2.com'
>>> dr2.name = 'company2'
>>> dr2.mail = 'contact@company2'
>>> dr2.phonenumber = '56789'
>>> dr2.save()
```

RELATIONSHIP OF ORM:

Object-Relational Mapping (ORM), relationships refer to how different database tables or models are related or connected to each other. These relationships define how data in one table or model is associated with data in another table or model. There are typically three primary types of relationships in ORM:

One-to-One (1:1) Relationship:

In a one-to-one relationship, each record in one table or model is associated with exactly one record in another table or model, and vice versa. It's like having a direct link between two entities.

For example, consider a scenario where you have a "Person" model and a "Passport" model. Each person can have only one passport, and each passport belongs to only one person.

```
from django.db import models

class Vehicle(models.Model):
    reg_no = models.IntegerField()
    owner = models.CharField(max_length = 100)

class Car(models.Model):
    vehicle = models.OneToOneField(Vehicle,
          on_delete = models.CASCADE, primary_key = True)
    car_model = models.CharField(max_length = 100)
```

This is used when one record of a model A is related to exactly one record of another model B. This field can be useful as a primary key of an object if that object extends another object in some way. For example – a model Car has one-to-one relationship with a model Vehicle, i.e. a car is a vehicle. One-to-one relations are defined using OneToOneField field of django.db.models.

One-to-Many (1:N) Relationship:

In a one-to-many relationship, each record in one table or model can be associated with multiple records in another table or model, but each record in the second table is associated with only one record in the first table.

A common example is a "User" model having multiple "Posts." Each user can have many posts, but each post belongs to only one user.

```
from django.db import models

class Album(models.Model):
    title = models.CharField(max_length = 100)
    artist = models.CharField(max_length = 100)

class Song(models.Model):
    title = models.CharField(max_length = 100)
    album = models.ForeignKey(Album, on_delete = models.CASCADE)
```

This is used when one record of a model A is related to multiple records of another model B. For example – a model Song has many-to-one relationship with a model Album, i.e. an album can have many songs, but one song cannot be part of multiple albums. Many-to-one relations are defined using ForeignKey field of django.db.models.

Many-to-Many (N:N) Relationship:

In a many-to-many relationship, multiple records in one table or model can be associated with multiple records in another table or model, and vice versa. This type of relationship often requires a junction table or intermediary model.

For instance, in a music streaming service, you could have a "User" model and a "Playlist" model. Users can have many playlists, and each playlist can be associated with multiple users. To handle this, you might use an intermediary table to represent which users have access to which playlists.

In an ORM framework like Django (for Python) or Hibernate (for Java), these relationships are defined using field types and relationships within the model classes, making it easier to work with databases without writing raw SQL queries.

```
from django.db import models

class Author(models.Model):
   name = models.CharField(max_length = 100)
   desc = models.TextField(max_length = 300)

class Book(models.Model):
   title = models.CharField(max_length = 100)
   desc = models.TextField(max_length = 300)
   authors = models.ManyToManyField(Author)
```

This is used when one record of a model A is related to multiple records of another model B and vice versa. For example – a model Book has many-to-many relationship with a model Author, i.e. an book can be written by multiple authors and an author can write multiple books. Many-to-many relations are defined using ManyToManyField field of django.db.models.

Here's a brief summary of how these relationships are represented in Django:

One-to-One: Use OneToOneField.

One-to-Many: Use ForeignKey in the "many" side of the relationship.

Many-to-Many: Use ManyToManyField.

Understanding and correctly defining these relationships is crucial for designing a well-structured database and efficiently retrieving data in your application.

Model inheritance:

In Django, you can use model inheritance to create a new model that inherits fields and behavior from an existing model. This concept is similar to the way classes can inherit from other classes in object-oriented programming. Django provides three types of model inheritance

These three types of model inheritance in Django allow you to reuse and extend existing models, making your database design more modular and maintaining a DRY (Don't Repeat Yourself) codebase. Each type has its own use case, so choose the one that best fits your requirements.

1.Abstract Base Classes:

Abstract base classes are used to define common fields and methods shared by multiple models. They do not create database tables on their own. Instead, they serve as a blueprint for other models.

You define an abstract base class by setting abstract = True in the model's Meta class. Other models can then inherit from this abstract base class using the abstract = True option in their own Meta class.

#models.py

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    roll_number = models.CharField(max_length=10)
```

2.Multi-Table Inheritance:

Multi-table inheritance creates a one-to-one relationship between a parent model and a child model. Each model gets its own database table, but the child model can inherit fields and methods from the parent model.

This type of inheritance is useful when you want to add specific fields or methods to a model that inherits from another model.

#models.py

Example:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=100)

class Restaurant(Place):
    serves_pizza = models.BooleanField(default=False)
```

Proxy Models:

Proxy models allow you to create a different Python class that represents the same database table as an existing model. Proxy models are read-only, and they don't create a new database table.

Proxy models are useful for adding new methods, custom managers, or changing the behavior of an existing model without modifying the original model

#models.py

Example:

```
from django.db import models

class MyModel(models.Model):
    name = models.CharField(max_length=50)

class MyModelProxy(MyModel):
    class Meta:
        proxy = True

    def custom_method(self):
        return f"Hello, {self.name}!"
```

CRUD OPERATIONS:

#models.py

CRUD operations stand for Create, Read, Update, and Delete, which are the basic operations for managing data in a database or through an Object-Relational Mapping (ORM) framework like Django.

These CRUD operations form the core of database interactions in Django's ORM. You can use them to create, retrieve, update, and delete data in your application's database while abstracting away the underlying SQL queries. Additionally, Django's ORM provides a powerful and flexible way to work with your data, and it includes features like model methods, database migrations, and query expressions to make these operations more efficient and expressive.

Create (C):

Creating records in the database. In Django, this is done using the model's constructor (e.g., MyModel.objects.create(...)) or by creating an instance of the model and then calling the save() method.

```
# Example of creating a new record
from myapp.models import MyModel

# Using the constructor
MyModel.objects.create(field1='value1', field2='value2')

# Using an instance and then saving it
new_instance = MyModel(field1='value1', field2='value2')
new_instance.save()
```

Read (R):

Reading or retrieving data from the database. In Django, you can use the model's manager to query for records, apply filters, and retrieve data.

```
# Example of reading data
from myapp.models import MyModel

# Retrieve all records
all_records = MyModel.objects.all()

# Retrieve records that meet specific criteria
filtered_records = MyModel.objects.filter(field1='value1')
```

Update (U):

Modifying existing records in the database. You can use the model's manager to retrieve records, change their values, and then call the save() method to update the database.

```
# Example of updating data
from myapp.models import MyModel

# Retrieve a record
record_to_update = MyModel.objects.get(id=1)

# Update the record
record_to_update.field1 = 'new_value'
record_to_update.save()
```

Delete (D):

Deleting records from the database. In Django, you can use the model's manager to query for records and then call the delete() method on the queryset to remove them from the database.

```
# Example of deleting data
from myapp.models import MyModel

# Retrieve a record and delete it
record_to_delete = MyModel.objects.get(id=1)
record_to_delete.delete()
```

ADMIN :

Django Admin is a really great tool in Django, it is actually a CRUD* user interface of all your models!

Django Admin is a powerful, built-in feature of the Django web framework that provides an easy-to-use and customizable administration interface for managing the data in your Django application's database. It's a web-based tool that allows authorized users to perform various administrative tasks without having to write custom views, templates, or forms. Here are some key points and features of Django Admin

Create User:

```
>> python manage.py createsuperuser
```

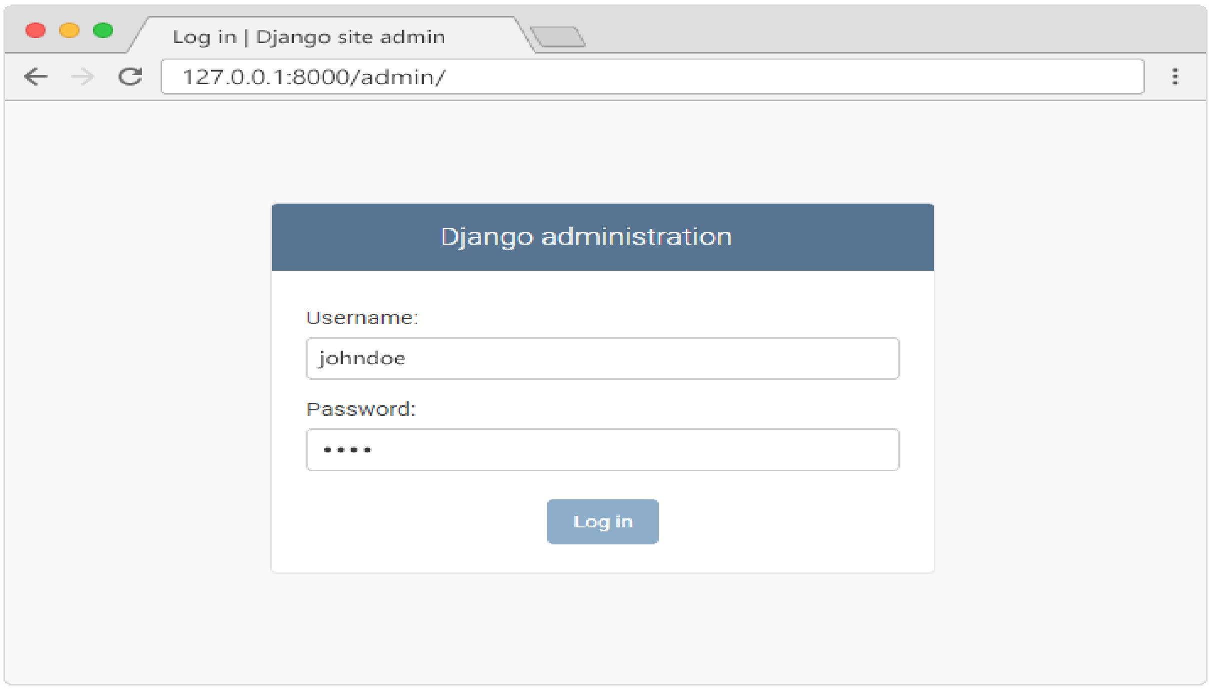
```
# Hit Enter

Username: johndoe
Email address: johndoe@dummymail.com
Password:
Password (again):
```

Superuser created successfully

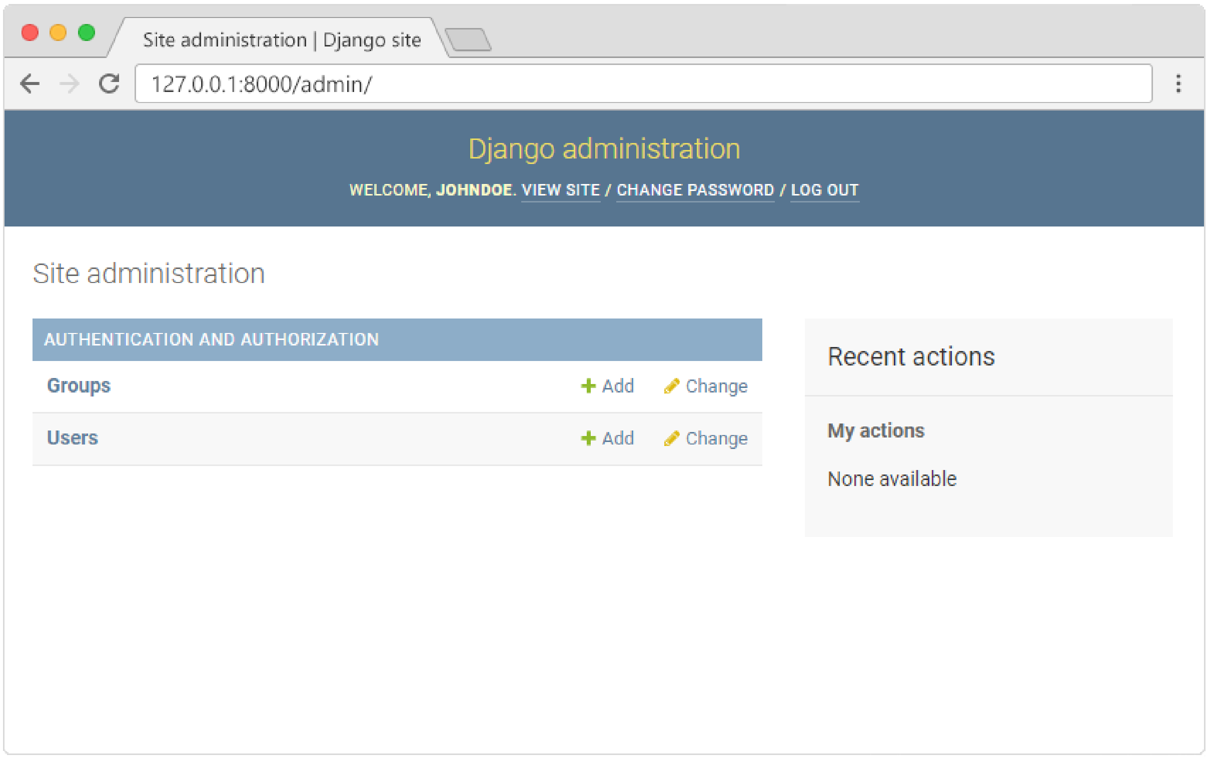
Now start the server again:

```
python manage.py runserver
```

Below diagram shows the admin page:



ADMIN PANEL:


Site administration | Django site

127.0.0.1:8000/admin/

Django administration

WELCOME, **JOHNDOE**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

Recent actions

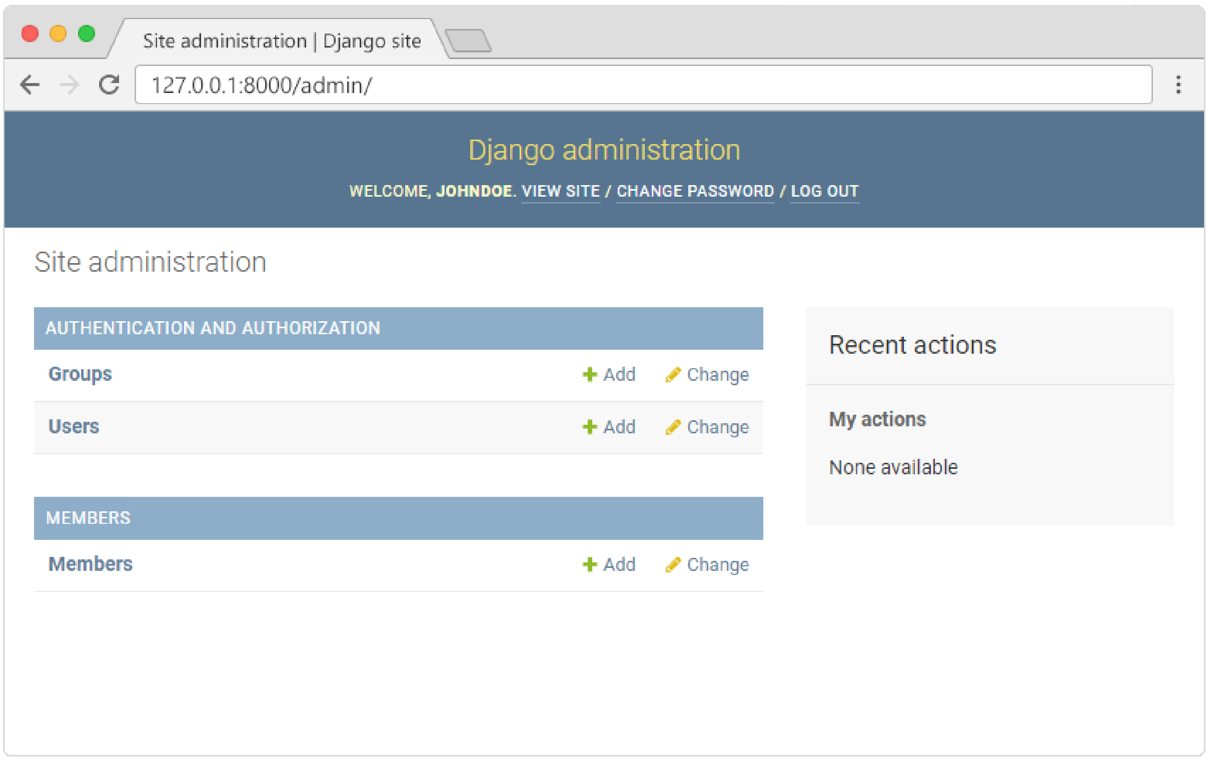
My actions

None available

Include models in the Admin Interface:

We need to configure the model details on admin.py

```
from .models import Members  #classname


# Register your models here.

admin.site.register(Members)
```


Site administration | Django site

127.0.0.1:8000/admin/

Django administration

WELCOME, JOHNDOE. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups + Add Change

Users + Add Change

MEMBERS

Members + Add Change

Recent actions

My actions

None available

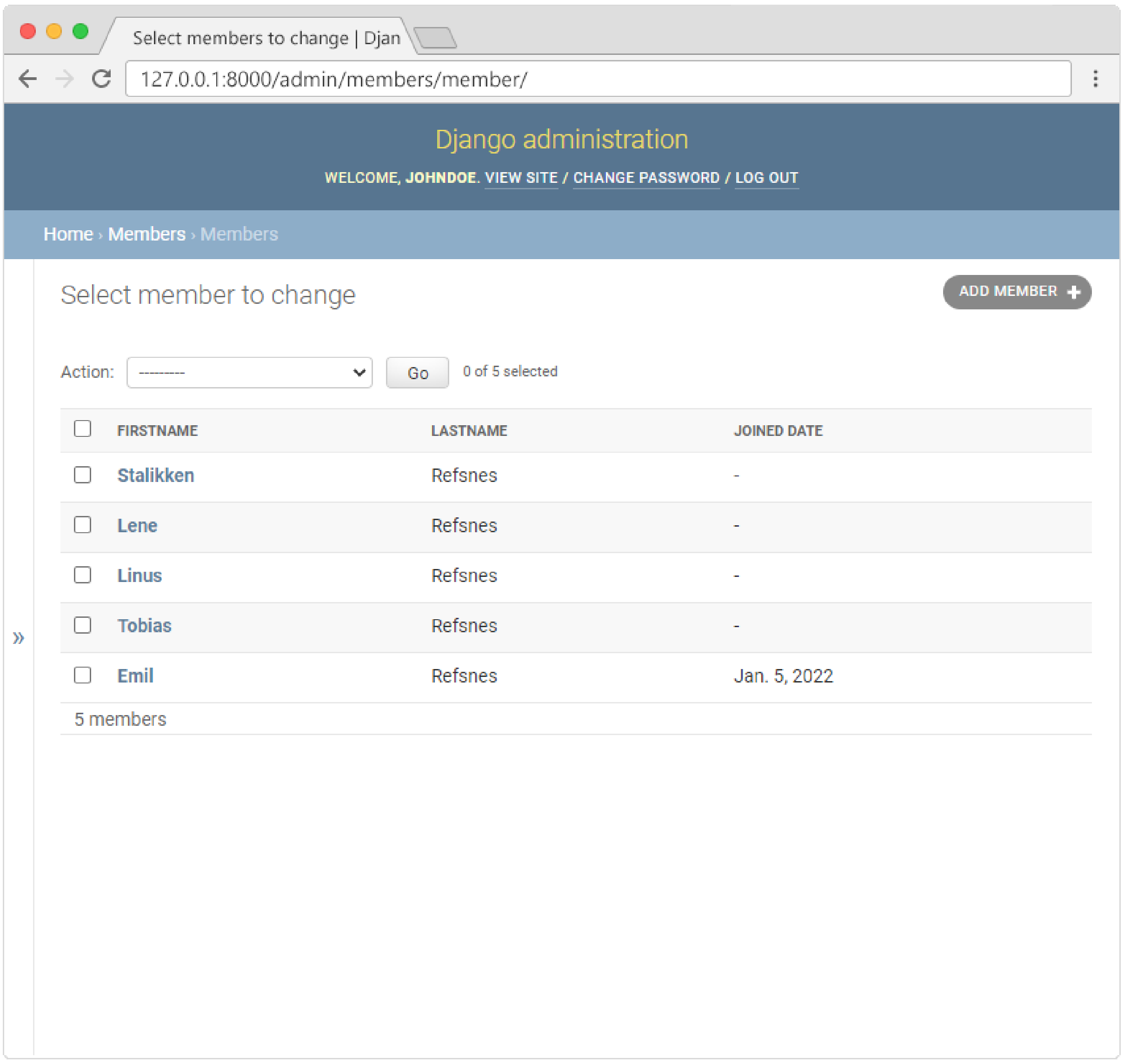
ADMIN PAGE COUSTMIZATION:

We can control the fields to display by specifying them in in a list_display property in the admin.py file.

First create a MemberAdmin() class and specify the list_display tuple, like this:

```
#admin.py

from .models import Member

# Register your models here.

class MemberAdmin(admin.ModelAdmin):

  list_display = ("firstname", "lastname", "joined_date",)



admin.site.register(Member, MemberAdmin)
```

#after customization


Select members to change | Djan
127.0.0.1:8000/admin/members/member/
Django administration
WELCOME, JOHNDOE. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Members › Members
ADD MEMBER
Action:
Go

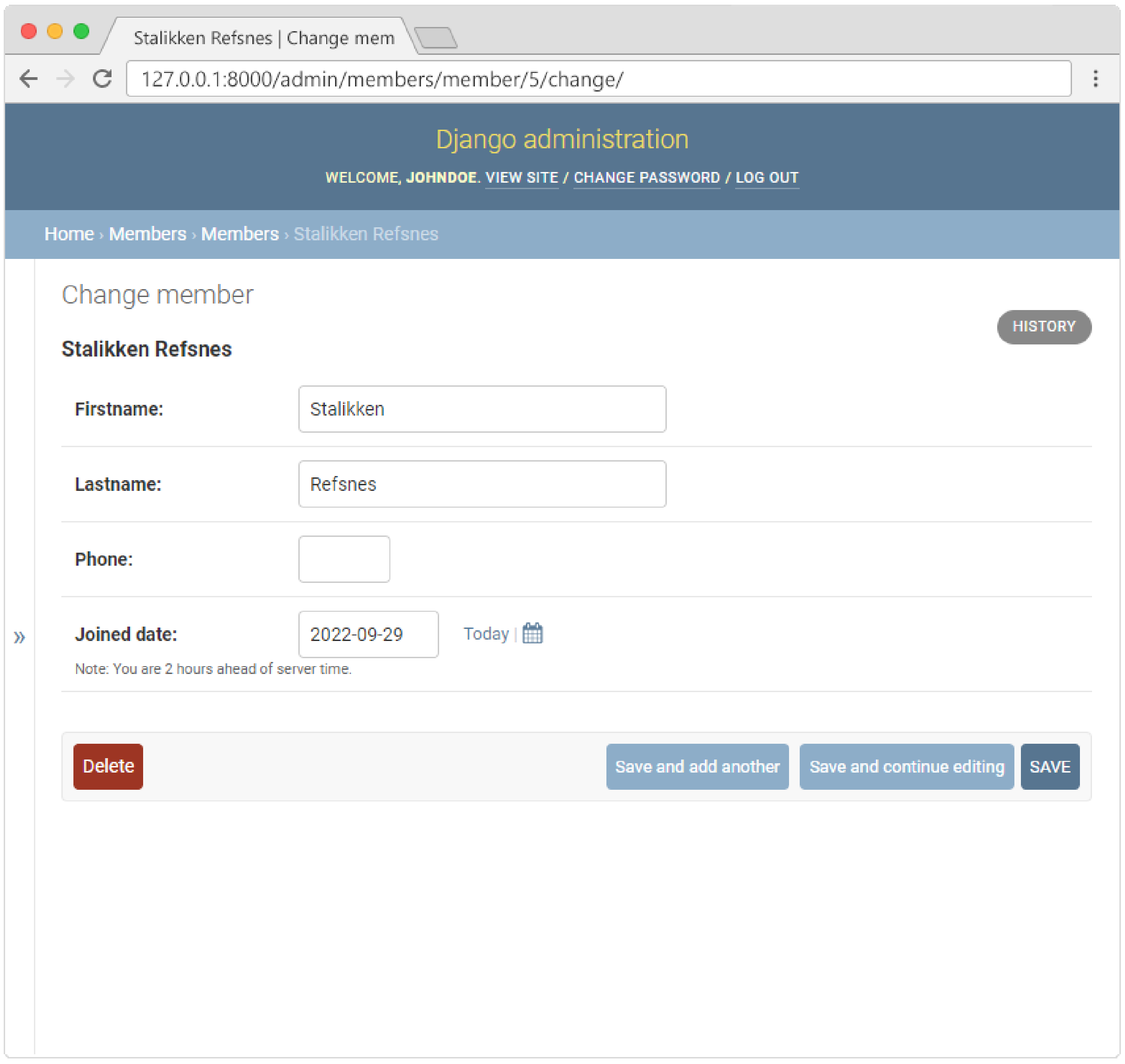
Select member to change

0 of 5 selected

FIRSTNAME	LASTNAME	JOINED DATE
Stalikken	Refsnes	-
Lene	Refsnes	-
Linus	Refsnes	-
Tobias	Refsnes	-
Emil	Refsnes	Jan. 5, 2022

5 members

Now we are able to create, update, and delete members in our database, and we start by giving them all a date for when they became members.


Stalikken Refsnes | Change mem

127.0.0.1:8000/admin/members/member/5/change/

Django administration

WELCOME, JOHNDOE. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Members › Members › Stalikken Refsnes

Change member

Stalikken Refsnes

HISTORY

Firstname: Stalikken

Lastname: Refsnes

Phone:

Joined date: 2022-09-29 Today |

Note: You are 2 hours ahead of server time.

Delete | Save and add another | Save and continue editing | SAVE

DJANGO DEBUG TOOLBAR:

What is Django Debug Toolbar?

The Django Debug Toolbar is a configurable set of panels that display various debug information about the current request or response and, when clicked, display more details about the panel's content.
For installation, I recommend using the virtual environment and executing this command:

```
$ pip install django-debug-toolbar
```

The configuration of this component is simple; you just need to change your project's urls.py and settings.py to use the toolbar. In your project's urls.py, enter this code:

```
from django.conf import settings

from django.urls import include, path  # For django versions from 2.0 and up

import debug_toolbarurlpatterns = [

    #...

    path('__debug__/', include(debug_toolbar.urls)),

    #...



]
```

Now In your project's settings.py, make sure that debug mode is true.

```
DEBUG = True
```

Add debug_toolbar and check that django.contrib.staticfiles is present in INSTALLED_APPS.

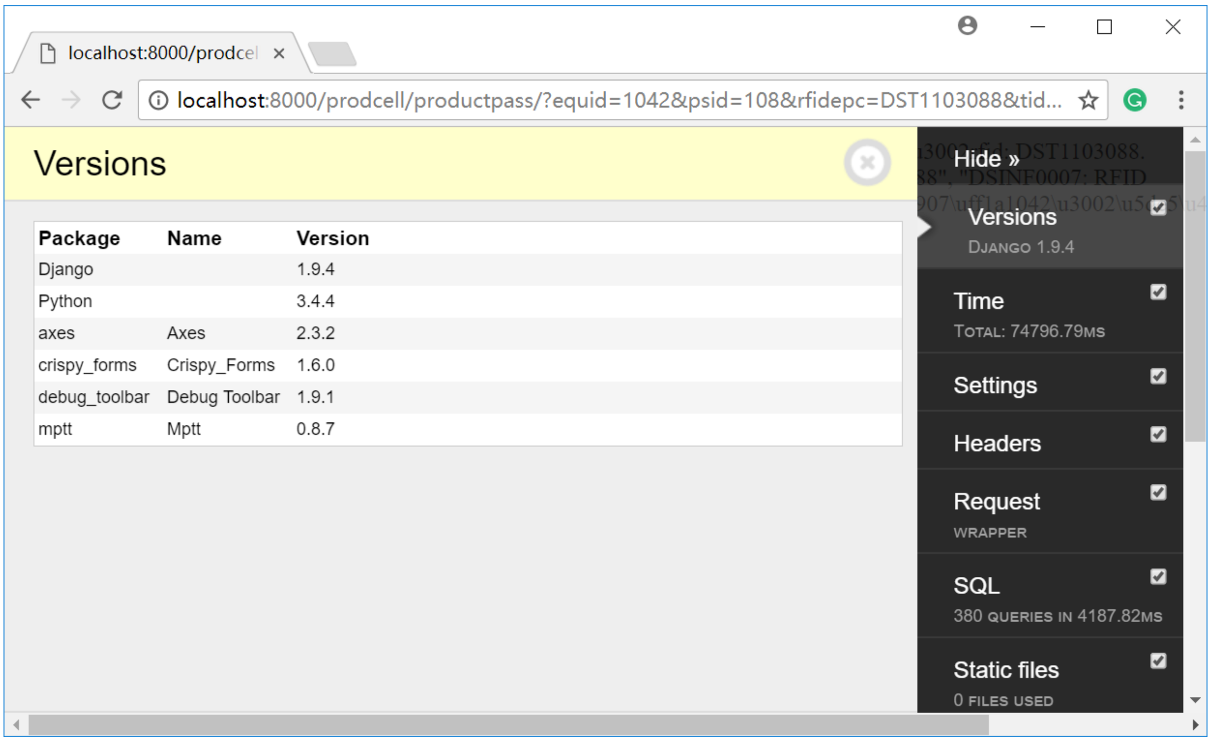
```
INSTALLED_APPS = [
#...
'django.contrib.staticfiles',
#...
'debug_toolbar',
#...
]
```

Add this line to MIDDLEWARE

```
MIDDLEWARE = [
#...
'debug_toolbar.middleware.DebugToolbarMiddleware',
#...
]
```

Add INTERNAL_IPS in settings.py. The INTERNAL_IPS configuration is valid for a local development environment; if your dev environment is different, you just change this field with your valid configuration.

```
INTERNAL_IPS = ('127.0.0.1', '0.0.0.0', 'localhost',)
```



Versions

Package	Name	Version
Django		1.9.4
Python		3.4.4
axes	Axes	2.3.2
crispy_forms	Crispy_Forms	1.6.0
debug_toolbar	Debug Toolbar	1.9.1
mptt	Mptt	0.8.7

MIDDLEWARES:

Middlewares in Django are a fundamental part of the request/response processing pipeline in a Django web application. They are designed to perform various tasks and functions at different stages of the request handling process

Processing Requests:

Middlewares can preprocess and modify incoming HTTP requests before they reach the view functions. This allows you to perform tasks like authentication, request parsing, and security checks.

Request-Response Transformation:

Middlewares can manipulate the request or the response, altering their content, headers, or other attributes. For example, you can compress response content, set cache headers, or modify cookies.

Authentication and Authorization:

Middleware can handle user authentication and authorization, ensuring that only authorized users can access specific parts of your site or perform certain actions.

Security Enhancements:

You can use middleware to add security measures such as Cross-Site Request Forgery (CSRF) protection, clickjacking prevention, and content security policies to enhance the security of your web application.

Logging and Profiling:

Middleware can log requests and responses, which is useful for monitoring and diagnosing issues in your application. It can also be used for profiling to analyze application performance.

Localization and Internationalization:

Middleware can handle localization and internationalization tasks, such as setting the language and timezone based on user preferences or browser settings.

User Agent and Device Detection:

Middlewares can detect the user's device type (desktop, mobile, tablet) or user agent and adapt the response accordingly, providing a responsive design.

Caching:

Middlewares can manage caching mechanisms, including browser cache headers and server-side caching to improve performance.

Custom Behavior:

You can implement custom behaviors and actions that need to be executed on every request or response. This can include adding custom headers or processing specific types of requests.

Error Handling:

Middleware can capture and handle exceptions or errors that occur during request processing. This is useful for providing custom error pages or for reporting errors.

Redirects and URL Routing:

Middlewares can perform URL redirection based on certain conditions, such as redirecting to a different URL or modifying the URL before the view is executed.

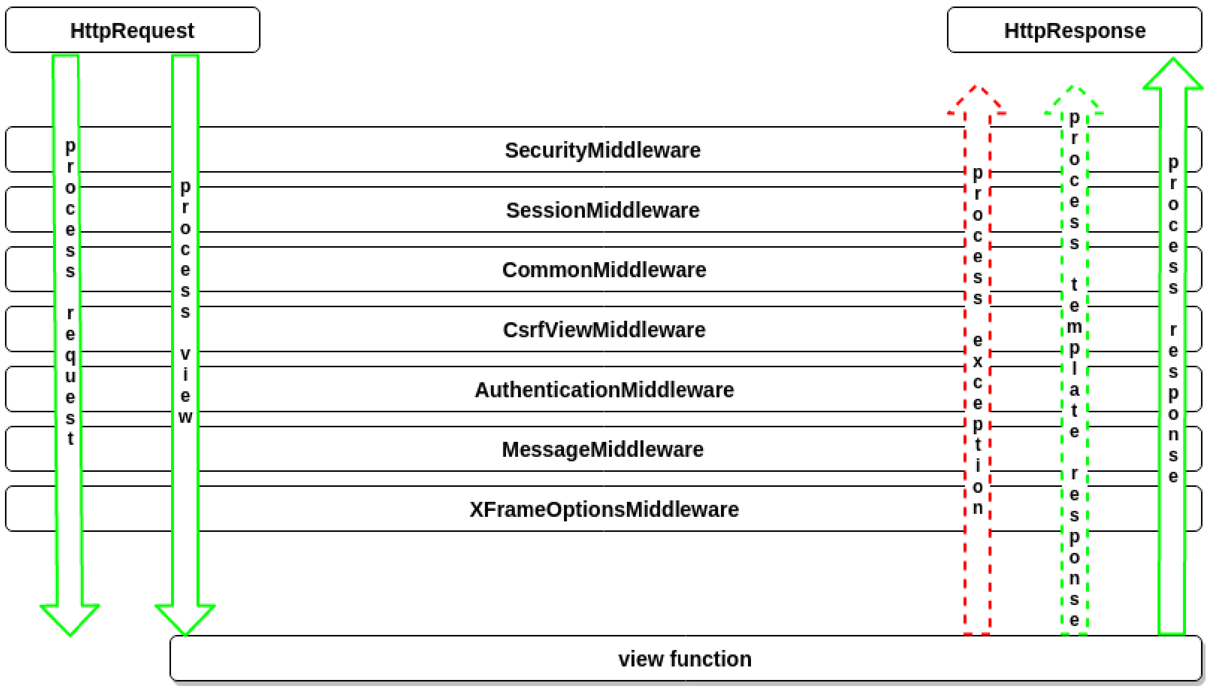
Session Management:

Middleware can handle session management, allowing you to store and retrieve session data for authenticated users.

Django Rest Framework Integration:

If you're building a RESTful API using Django Rest Framework, middlewares can be used to handle authentication, token validation, and request parsing specific to APIs.

In Django, you can create and configure custom middleware classes to perform these tasks and more. Middleware classes are executed in the order they are defined in the MIDDLEWARE setting, allowing you to control the order in which middleware functions are applied to requests and responses. This flexibility enables you to tailor your application's behavior to your specific needs and requirements.



Forms

What are forms?

In web development, a "form" refers to an essential HTML element that allows users to submit data to a web server. A form typically contains various input fields (such as text inputs, checkboxes, radio buttons, and dropdowns) where users can enter or select information. After filling out the form, users can click a "Submit" button to send their data to the server for further processing.

Forms are a fundamental part of web applications and websites, enabling various interactions and functionalities, such as user registration, login, search, data submission, and much more. They serve as a means of communication between the user and the server, facilitating data transfer and user engagement.

Why are forms important in web development?

- Forms are crucial in web development for several reasons:
- **User Interaction:** Forms allow users to interact with websites and web applications. Users can provide input, submit data, and make selections, making it possible to perform actions such as signing up, logging in, making purchases, and providing feedback.
- **Data Collection:** Websites and applications often require user-provided data, such as usernames, passwords, addresses, and search queries. Forms are the primary means to collect this information.

- **Data Submission:** Forms enable the submission of data to web servers, where it can be processed, stored, or used for various purposes. For instance, a user's login credentials are submitted via a form to authenticate them.
- **User Feedback:** Forms can be used to collect feedback and opinions from users through surveys, contact forms, and comment sections, enhancing user engagement and user experience.
- **User Registration:** User registration forms are used to create user accounts, store user information, and grant access to personalized content and features.
- **E-commerce:** Forms are crucial in e-commerce for product searches, shopping carts, and payment information collection.
- **Search Functionality:** Search forms allow users to enter keywords and search for specific content within a website or application.
- **Data Validation:** Forms often include validation rules to ensure that the data submitted is accurate and meets specific criteria. This helps maintain data quality and security.
- **Security:** Forms play a vital role in security, especially in protecting against cross-site request forgery (CSRF) attacks. Security measures like CSRF tokens are integrated into forms to prevent unauthorized form submissions.

- **10. User Experience:** Well-designed forms enhance the user experience by providing clear instructions, error handling, and a structured way to provide information or take action.

HTML Forms

1. **Basic HTML Form Structure**

```
<!DOCTYPE html>

<html>

<head>

  <title>Sample Form</title>

</head>

<body>

  <form action="/submit" method="post">

    <label for="username">Username:</label>

    <input type="text" id="username" name="username" required>

    <label for="password">Password:</label>

    <input type="password" id="password" name="password" required>

    <input type="submit" value="Submit">

  </form>

</body>

</html>
```

2. **Form Elements**

• Text Input

```
<input type="text" name="name" placeholder="Your Name">
```

• Textarea

```
<textarea name="message" rows="4" cols="50">Enter your message here.</textarea>
```

• Select Dropdown

```
<select name="country">
    <option value="us">United States</option>
    <option value="ca">Canada</option>
    <option value="uk">United Kingdom</option>
</select>
```

• Form Attributes

```
<form action="/submit" method="post" enctype="multipart/form-data">
    <!-- Form elements go here -->
</form>
```

Handling Form Submissions

GET Request

```
<form action="/search" method="get">
    <input type="text" name="query" placeholder="Search">
    <input type="submit" value="Search">
</form>
```

POST Request

```
<form action="/login" method="post">
    <input type="text" name="username" placeholder="Username">
    <input type="password" name="password" placeholder="Password">
    <input type="submit" value="Login">
</form>
```

Django Forms

What are Django Forms?

Django forms are Python classes that simplify form handling. Here's an example:

```
from django import forms
class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
```

Benefits of Using Django Forms

Django forms provide automatic validation and error handling. In a view:

```
def contact_view(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Process the valid form data
        else:
            # Handle form errors
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

Form Fields

Django form fields define what data is collected. Here are some examples:

```
from django import forms
class LoginForm(forms.Form):
    username = forms.CharField(max_length=100, label="Username")
    password = forms.CharField(widget=forms.PasswordInput)
class RegistrationForm(forms.Form):
    username = forms.CharField(max_length=100)
    email = forms.EmailField()
    password = forms.CharField(widget=forms.PasswordInput
```

These examples illustrate the basic concepts of HTML forms and Django forms with simple code snippets. You can build upon these examples for more complex use cases in your web development projects.

Form Widgets

What are Form Widgets?

Form widgets are an integral part of HTML forms and are used to define how data is presented and collected in a web form. They determine the appearance and interactivity of form elements. In Django, form widgets are associated with form fields, and they allow you to customize the way data is displayed and input by users.

Different Types of Form Widgets in Django

Django provides a variety of built-in form widgets for different types of form fields. Some common form widgets include:

- TextInput: A single-line text input.

- PasswordInput: A text input for password fields that hides the entered characters.

- EmailInput: A text input with email validation.

- Textarea: A multi-line text input.

- CheckboxInput: A checkbox for boolean (True/False) values.

- Select: A dropdown select field.

- RadioSelect: Radio buttons for selecting one option from a list.

- CheckboxSelectMultiple: Checkboxes for selecting multiple options from a list.

How to Customize Form Widgets

You can customize form widgets in Django by specifying them in the form field's widget attribute. Here's an example of how to customize a form field's widget:

```
from django import forms

class MyForm(forms.Form):

    name = forms.CharField(

        max_length=100,

        widget=forms.TextInput(attrs={'placeholder': 'Enter your name'})

    )

    password = forms.CharField(

        widget=forms.PasswordInput(attrs={'placeholder': 'Enter your password'})

    )
```

Rendering Forms in Templates

Displaying Forms in HTML Templates

Django provides several methods to render forms in HTML templates, making it easy to integrate forms into your web pages. You can use the following template tags to render forms:

- {{ form.as_p }}: Renders the form as paragraphs.

- {{ form.as_table }}: Renders the form as an HTML table.

- {{ form.as_ul }}: Renders the form as list items.

- Manual Rendering: You can manually render form fields using {{ form.field_name }} or loop through form fields in the template.

Example of Rendering Forms in Templates

Suppose you have a simple Django form called ContactForm with fields for name, email, and message. You can render this form in a template as follows:

```
<form method="post" action="{% url 'contact' %}">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Submit</button>
</form>
```

Handling Form Errors and Displaying Error Messages

To handle form errors and display error messages in the template, you can use the following template tags and filters:

- {% for error in form.field_name.errors %}: Iterates through field-specific errors.

- {{ form.field_name.errors }}: Displays all errors for a field.

- {{ form.non_field_errors }}: Displays non-field-specific errors (e.g., form-wide errors).

For example, to display errors for the email field in the ContactForm, you can use:

```
{% for error in form.email.errors %}
    <div class="error">{{ error }}</div>
{% endfor %}
```

Form Validation

Automatic Data Validation in Django Forms

Django provides automatic data validation for forms. When you call form.is_valid() in your view, Django checks the submitted data against the form's defined fields, including their types and any specified validation rules (e.g., required fields, maximum length). If the data doesn't meet these criteria, the form is considered invalid, and you can access the validation errors to display them to the user.

Example:

```
from django import forms
class ContactForm(forms.Form):
    name = forms.CharField(max_length=100)
    email = forms.EmailField()
    message = forms.CharField(widget=forms.Textarea)
# In a view
def contact_view(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            # Data is valid, process it
        else:
            # Data is invalid, handle errors
    else:
        form = ContactForm()
    return render(request, 'contact.html', {'form': form})
```

Writing Custom Validation Logic in Form Classes

You can add custom validation methods to your form class by creating methods that follow the naming convention clean_fieldname. For example, to validate the email field in a ContactForm, you can add a clean_email method:

```
from django import forms
from django.core.exceptions import ValidationError
class ContactForm(forms.Form):
    def clean_email(self):
        email = self.cleaned_data['email']
        if not email.endswith('@example.com'):
            raise ValidationError("Email must be from example.com domain.")
        return email
```

Form Field Validation and Global Form Validation

Django forms provide a way to validate individual form fields using clean_fieldname methods, as shown in the previous example. Additionally, you can perform global form-level validation by adding a clean method to your form class. The clean method is called after field-specific validation and allows you to validate relationships between fields or check for errors that involve multiple fields.

Example:

```
from django import forms
from django.core.exceptions import ValidationError
class OrderForm(forms.Form):
    quantity = forms.IntegerField()
    unit_price = forms.DecimalField()
    def clean(self):
        cleaned_data = super().clean()
        quantity = cleaned_data.get('quantity')
        unit_price = cleaned_data.get('unit_price')

        if quantity and unit_price and quantity * unit_price > 1000:
```

```
        raise ValidationError("Order value cannot exceed $1000.")
```

Handling Form Submissions

Receiving and Processing Form Data in Django Views

In Django views, you can receive and process form data by checking the request method (GET or POST) and working with the form instance. Here's an example of handling a POST request and processing form data:

Example:

```
from django.shortcuts import render

from django.http import HttpResponseRedirect

from .forms import ContactForm

def contact_view(request):

    if request.method == 'POST':

        form = ContactForm(request.POST)

        if form.is_valid():

            # Process the valid form data (e.g., send an email)

            return HttpResponseRedirect('/thank-you/')

    else:

        form = ContactForm()

    return render(request, 'contact.html', {'form': form})
```

Using `request.POST` to Access Form Data

In the view function, you can access form data using `request.POST`. This dictionary-like object contains the submitted data, with keys corresponding to the field names. For example:

Example:

```
name = request.POST.get('name')

email = request.POST.get('email')

message = request.POST.get('message')
```

Handling GET and POST Requests

Django allows you to handle both GET and POST requests in a view. Use `request.method` to determine the request type and take appropriate actions. GET requests are typically used to retrieve data, while POST requests are used for data submission and modification.

Example:

```
if request.method == 'GET':
    # Handle GET request
elif request.method == 'POST':
    # Handle POST request
```

CSRF Protection with `{% csrf_token %}`

To protect against Cross-Site Request Forgery (CSRF) attacks, include `{% csrf_token %}` within your HTML form. This template tag adds a hidden field containing a unique token, which Django uses to validate the form submission.

Example:

```
<form method="post" action="{% url 'submit_form' %}">
    {% csrf_token %}
    <!-- Form fields go here -->
    <input type="submit" value="Submit">
</form>
```

Form Submissions and Redirection

Redirecting Users After a Successful Form Submission

After processing a valid form submission, it's common to redirect users to a thank-you page or another relevant page. You can use `HttpResponseRedirect` to achieve this:

Example:

```
from django.http import HttpResponseRedirect
def submit_form(request):
    if request.method == 'POST':
        # Process the form data
        # Redirect to a thank-you page
        return HttpResponseRedirect('/thank-you/')
```

Handling Form Submission Errors

If the form submission contains errors, you can handle them in various ways, such as re-rendering the form with error messages or redirecting users to an error page, depending on your application's design.

Using Django's `HttpResponseRedirect`

`HttpResponseRedirect` is a Django class that creates an HTTP response for redirecting users to a different URL. You provide the target URL as an argument, and Django will issue an HTTP redirect response to the client's browser, which will navigate the user to the specified URL.

Example:

```
from django.http import HttpResponseRedirect

def submit_form(request):

    if request.method == 'POST':

        # Process the form data

        # Redirect to a thank-you page

        return HttpResponseRedirect('/thank-you/')
```

This code redirects users to the `/thank-you/` URL after a successful form submission. You can customize the target URL as needed.

Model Forms

Creating Forms from Django Models

Django provides a feature called "Model Forms" that allows you to create forms directly from existing Django models. This simplifies the process of creating forms for database models, as it automatically generates form fields based on the model's fields.

Example:

Suppose you have a model named `Product`:

```
from django.db import models

class Product(models.Model):

    name = models.CharField(max_length=100)

    description = models.TextField()

    price = models.DecimalField(max_digits=5, decimal_places=2)
```

You can create a form from this model as follows:

```
from django import forms

from .models import Product

class ProductForm(forms.ModelForm):

    class Meta:

        model = Product

        fields = '_all_'
```

Automatic Form Generation from Model Fields

By setting `fields = '__all__'` in the form's `Meta` class, the `ProductForm` will automatically generate fields corresponding to the model fields (`name`, `description`, `price`). This reduces the need for manually defining form fields.

Saving Form Data to the Database

To save form data to the database, you can call the `save()` method on a valid model form instance. This method automatically creates or updates a model instance based on the form data.

Example:

```
if request.method == 'POST':
    form = ProductForm(request.POST)
    if form.is_valid():
        product = form.save()  # Save the form data as a new or updated product
```

File Uploads

Handling File Uploads in Django Forms

Django makes it easy to handle file uploads in forms using the `FileField` and `ImageField` form fields. These fields allow users to upload files (e.g., images, documents) as part of a form submission.

Example:

```
from django import forms
class DocumentForm(forms.Form):
    docfile = forms.FileField(label='Select a file')
```

Using `FileField` and `ImageField`

- `FileField`: Used for general file uploads.

- `ImageField`: A specialized version of `FileField` for image uploads. It automatically validates that the uploaded file is an image.

Storing and Retrieving Uploaded Files

Django's default behavior is to store uploaded files in the local file system. You can configure the storage location in your project's settings. Django also provides built-in file handling features like serving files, cleaning up old files, and generating file URLs.

Example:

```
from django.conf import settings
from django.core.files.storage import FileSystemStorage
fs = FileSystemStorage(location=settings.MEDIA_ROOT)
```

```
filename = fs.save('uploaded_files/myfile.txt', content)
```

Formsets

What are Formsets in Django?

Formsets are a way to work with multiple forms on a single page. They are useful when you want to handle a collection of forms that represent similar data or when you need to manage dynamic forms. Formsets allow you to create, update, and delete multiple instances of a model in a single operation.

Using `formset_factory` to Create Formsets

You can create a formset in Django using the `formset_factory` function. This function generates a formset class based on a form class.

Example:

```
from django.forms import formset_factory

from .forms import MyForm

MyFormSet = formset_factory(MyForm, extra=2)
```

In this example, we create a formset class called `MyFormSet` based on the `MyForm` class. The `extra` parameter specifies the number of empty forms to display in the formset.

Working with Multiple Forms on a Single Page

To work with multiple forms on a single page, you typically loop through the formset in your template to render each form.

Example in a Django template:

```
<form method="post">

  {% csrf_token %}

  {% for form in formset %}

    {{ form.as_p }}

  {% endfor %}

  <input type="submit" value="Submit">

</form>
```

Advanced Topics

Working with Form Data Outside of Views (Context Processors)

Django provides context processors, which are functions that add data to the context of every template rendering. You can use context processors to make form data available in all templates without having to pass it explicitly in each view.

Example context processor:

```
def common_data(request):
    # Add form data to the context
    return {'my_form': MyForm()}
```

Form Customization Using Widgets, Labels, and `help_text`

You can customize form fields using various attributes:

- Widgets: Customize the HTML rendering of form fields (e.g., `TextInput(attrs={'class': 'custom-input'})`).
- Labels: Add labels to form fields to provide context for users (e.g., `forms.CharField(label='Name')`).
- `help_text`: Provide additional information or instructions for a form field (e.g., `forms.EmailField(help_text='Enter a valid email address.')`).

Handling Complex Form Relationships (ModelMultipleChoiceField, etc.)

Django supports complex form relationships, such as handling many-to-many or one-to-many relationships between models. You can use fields like `ModelMultipleChoiceField` to manage these relationships in forms.

Example:

```
from django import forms

class ProductForm(forms.ModelForm):

    # Define a ModelMultipleChoiceField for a many-to-many relationship

    categories = forms.ModelMultipleChoiceField(queryset=Category.objects.all())
```

This field allows users to select multiple categories for a product.

These advanced topics provide more flexibility and customization options when working with Django forms, allowing you to handle complex use cases and create rich user experiences.

Examples

Example 1: Regular HTML Form

Suppose you want to create a simple feedback form in HTML:

```
<!DOCTYPE html>

<html>

<head>

    <title>Feedback Form</title>

</head>

<body>

    <h1>Give Us Your Feedback</h1>

    <form action="/submit-feedback" method="post">

        <label for="name">Name:</label>

        <input type="text" id="name" name="name" required>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <label for="feedback">Feedback:</label>
        <textarea id="feedback" name="feedback" required></textarea>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Example 2: Django Model Form

Suppose you have a Django model called `Task` and you want to create a form to add tasks:

```
# forms.py
from django import forms
from .models import Task

class TaskForm(forms.ModelForm):
    class Meta:
        model = Task
        fields = ['title', 'description', 'due_date']
# views.py
from django.shortcuts import render, redirect
from .forms import TaskForm
def create_task(request):
    if request.method == 'POST':
        form = TaskForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('task_list')
    else:
        form = TaskForm()
    return render(request, 'create_task.html', {'form': form})
```

Sessions:

Introduction to Sessions

Sessions in web development refer to a mechanism for storing and maintaining user-specific data across multiple HTTP requests. They are essential for maintaining state and user authentication within web applications. Sessions allow the server to recognize a user and store data associated with that user's visit.

Django Sessions Overview

Django manages sessions through a session framework that uses a session backend to store and retrieve session data. The SESSION_ENGINE and SESSION_COOKIE_NAME settings play crucial roles in configuring how sessions are managed.

Setting Up Sessions in Django

To enable sessions in a Django project, you need to follow these steps:

- Add 'django.contrib.sessions.middleware.SessionMiddleware' to the MIDDLEWARE setting in your settings.py.
- Configure session settings in the same settings.py file, such as the SESSION_ENGINE, SESSION_COOKIE_NAME, and other options.

Here's an example of how to configure sessions in settings.py:

```
# settings.py

# Add 'django.contrib.sessions.middleware.SessionMiddleware' to MIDDLEWARE.

MIDDLEWARE = [

   # ...

   'django.contrib.sessions.middleware.SessionMiddleware',

   # ...

]

# Configure session settings

SESSION_ENGINE = 'django.contrib.sessions.backends.db'  # Use the database-backed session engine.

SESSION_COOKIE_NAME = 'my_session_cookie'  # Customize the session cookie name.
```

Session Backends

Django provides multiple session backends to store session data. You can choose a backend based on your project's needs:

- Database : Stores session data in the database.
- Cache : Stores session data in a caching system (e.g., Redis, Memcached).
- File-Based : Stores session data in files on the server.

Here's an example of how to configure different session backends in settings.py:

```
# settings.py
# Database-backed session
SESSION_ENGINE = 'django.contrib.sessions.backends.db'
# Cache-backed session (using Redis)
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
SESSION_CACHE_ALIAS = 'default'  # Define the cache alias to use.
# File-based session
SESSION_ENGINE = 'django.contrib.sessions.backends.file'
SESSION_FILE_PATH = '/path/to/session/files/'  # Specify the path to store session files.
```

Session Timeout and Expiry

In Django, you can control session timeout and expiry using the SESSION_COOKIE_AGE setting, which determines the session duration in seconds. The SESSION_SAVE_EVERY_REQUEST setting determines whether the session is saved on every request.

```
# settings.py
# Set session timeout to 15 minutes (900 seconds).
SESSION_COOKIE_AGE = 900
# Save the session data on every request.
SESSION_SAVE_EVERY_REQUEST = True
```

Using Sessions in Views

You can access and modify session data in your Django views using the request.session object. Here are some code examples:

```
# views.py
from django.shortcuts import render
from django.http import HttpResponse
def set_session_data(request):
    # Set session data
    request.session['user_name'] = 'John'
    return HttpResponse('Session data set.')
def get_session_data(request):
    # Get session data
    user_name = request.session.get('user_name', 'Guest')
    return HttpResponse(f'Hello, {user_name}.')
```

```
def delete_session_data(request):
    # Delete session data
    if 'user_name' in request.session:
        del request.session['user_name']
    return HttpResponse('Session data deleted.')
```

Session Middleware

Session middleware plays a crucial role in managing sessions in Django. It handles session creation, updates, and saving. The order of middleware matters, and SessionMiddleware should be placed after AuthenticationMiddleware for user authentication to work correctly.

```
# settings.py
MIDDLEWARE = [
    # ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    # ...
]
```

Session Security

To secure sessions in Django, consider the following best practices:

- Use the SESSION_COOKIE_SECURE setting to ensure cookies are only transmitted over HTTPS.

- Use the SESSION_COOKIE_HTTPONLY setting to prevent client-side JavaScript from accessing the session cookie.

```
# settings.py
# Enable secure cookies (requires HTTPS).
SESSION_COOKIE_SECURE = True
# Set the session cookie as HTTP-only.
SESSION_COOKIE_HTTPONLY = True
```

Session Management in Templates

You can use session data in Django templates by accessing request.session and displaying session variables in your HTML templates. Here's an example:

Example:

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Session Example</title>
</head>
<body>
    <p>Welcome, {{ request.session.user_name }}</p>
</body>
</html>
```

Session-Based User Authentication

Implementing basic user authentication using sessions involves setting session variables during login and clearing them during logout. Here's a simplified

example:

```
# views.py
from django.contrib.auth import authenticate, login, logout
from django.shortcuts import render, redirect
from django.http import HttpResponse
def user_login(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user:
            login(request, user)
            return redirect('dashboard')
    return render(request, 'login.html')


def user_logout(request):
    logout(request)
    return redirect('login')
```

Custom Session Data

You can store custom data in sessions using the request.session object. For example, you can set a user's preferred theme:

```
# views.py
```

```
def set_theme(request, theme_name):
    request.session['theme'] = theme_name
    return HttpResponse('Theme set.')
```

```
# template.html
<!DOCTYPE html>
<html>
<head>
    <title>Custom Theme</title>
    <link rel="stylesheet" href="{% static 'css/{{ request.session.theme }}.css' %}">
</head>
<body>
    <!-- Your content here -->
</body>
</html>
```

Authentication in Django:

Authentication is the process of verifying the identity of a user, ensuring that they are who they claim to be. In Django, authentication deals with user management, login, and session management.

Django User Authentication:

Django provides a built-in authentication system through the django.contrib.auth module. You can use this system to manage user accounts, login, and user sessions. Here's a step-by-step guide:

Setting up the Authentication System:

First, make sure you have django.contrib.auth in your INSTALLED_APPS in your Django project's settings.

```
# settings.py
INSTALLED_APPS = [
    # ...
    'django.contrib.auth',
    'django.contrib.contenttypes',
    # ...
]
```

User Registration:

To allow users to register, you need to create a user registration form and a view to handle user registration. Here's an example of how you can create a registration form:

```
# forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm
class UserRegistrationForm(UserCreationForm):
    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']
```

User Login:

Create a view for user login using Django's built-in authentication views, like LoginView or a custom view. To use a built-in view, add the following to your URL configuration:

```
# urls.py
from django.contrib.auth.views import LoginView
urlpatterns = [
    # ...
    path('login/', LoginView.as_view(), name='login'),
    # ...
]
```

User Logout:

For user logout, you can use the built-in LogoutView:

```
# urls.py
from django.contrib.auth.views import LogoutView

urlpatterns = [
    # ...
    path('logout/', LogoutView.as_view(), name='logout'),
    # ...
]
```

Authorization in Django:

Authorization is the process of determining what actions a user is allowed to perform after they are authenticated. Django provides a flexible and powerful authorization system.

Permissions and User Groups:

Django uses permissions and user groups to control authorization. You can create custom permissions and assign them to user groups or individual users. For

example:

```
from django.contrib.auth.models import Permission, User

# Create a custom permission

permission = Permission.objects.create(codename='can_publish', name='Can Publish Posts')

# Assign the permission to a user group

group = Group.objects.get(name='Editors')

group.permissions.add(permission)
```

Authorization in Views:

In views, you can use the @login_required decorator to ensure that only authenticated users can access a view. For more fine-grained control, you can use the @permission_required decorator.

Example:

```
from django.contrib.auth.decorators import login_required, permission_required

@login_required

def my_view(request):

    # This view can only be accessed by authenticated users

@permission_required('myapp.can_publish')

def publish_post(request):

    # This view can only be accessed by users with the 'can_publish' permission
```

Object-level Permissions:

Django also supports object-level permissions. You can specify who can edit, view, or delete a specific object in your application. This is done using the django.contrib.auth package and the django.contrib.contenttypes framework.

Here's an example of how you can use object-level permissions:

```
# models.py

from django.db import models

from django.contrib.auth.models import Permission

class MyModel(models.Model):
    # Your model fields
# Create a permission for MyModel
permission = Permission.objects.create(
    codename='can_edit_mymodel',
    name='Can Edit MyModel',
    content_type=ContentType.objects.get_for_model(MyModel),
)
```

Paginators

Creating a Paginator for E-commerce in Django

Introduction

Paginator is like a tool that helps us manage a long list of items by dividing it into smaller, more manageable pieces or pages. Imagine you have a big phone book with hundreds of names, and instead of flipping through the entire book, you get smaller sections with names. Paginator in Django works the same way; it helps break up a long list of items, like products in an online store, into pages so that users can easily navigate through them without overwhelming the webpage. It's a way to make your website more user-friendly and efficient, especially when dealing with a lot of data.

Following are the benefits:

- Improve Performance
- User -Friendly Navigation
- Enhanced Readability
- Optimized Resources Usage

Steps to follow to create pagination

Step 1: Create Views for Pagination

```
from django.core.paginator import Paginator
from django.shortcuts import render
from .models import Product    # import your model

def product_list(request):
    products = Product.objects.all()
    paginator = Paginator(products, 10)  # Show 10 products per page
    page_number = request.GET.get('page')
    page = paginator.get_page(page_number)
    return render(request, 'products/product_list.html', {'page': page})
```

Step 2: create URL for function

```
from django.urls import path
from . import views

urlpatterns = [
    path('products/', views.product_list, name='product_list'),
]
```

Step 3: Create a Template:

Design the HTML for Product listing with pagination

```
<!DOCTYPE html>
<html>
<head>
    <title>Product List</title>
</head>
<body>
    <h1>Product List</h1>

    <ul>
        {% for product in page %}
            <li>{{ product.title }} - {{ product.price }}</li>
        {% endfor %}
    </ul>


    <div class="pagination">
        <span class="step-links">
            {% if page.has_previous %}
                <a href="?page=1">&laquo; first</a>
                <a href="?page={{ page.previous_page_number }}">previous</a>
            {% endif %}


            <span class="current-page">{{ page.number }} of {{ page.paginator.num_pages
}}</span>


            {% if page.has_next %}
                <a href="?page={{ page.next_page_number }}">next</a>
                <a href="?page={{ page.paginator.num_pages }}">last &raquo;</a>
            {% endif %}
        </span>
    </div>
</body>
</html>
```

Here is the Example of Page :

example.com/shirt/page=1
< 1 2 3 4 ··· 9 >
example.com/shirt/page=5
< 1 ··· 4 5 6 ··· 9 >
example.com/shirt/page=9
< 1 ··· 6 7 8 9 >

Signals

Introduction to Django Signals

Django, a popular Python web framework, provides a powerful mechanism called "signals" for decoupling various components of your application. Django signals enable different parts of your application to communicate without direct dependencies, offering flexibility, modularity, and extensibility. They are used to allow certain senders to notify a set of receivers that some action has taken place.

Description

Django signals work on the publisher-subscriber pattern. In this pattern, the sender or "publisher" sends a signal when a particular event occurs, and one or more "subscribers" or receivers respond to that signal by executing specific functions or methods.

Common use cases for Django signals include:

Automated Actions: Triggering actions when certain events occur, such as sending a welcome email when a new user registers.

Model Preprocessing: Performing actions before or after a model is saved, deleted, or updated, like generating a slug for a new article.

Decoupled Applications: Allowing different apps within a Django project to interact without needing to be aware of each other's existence.

Logging and Monitoring: Logging and monitoring application events for debugging or performance analysis.

Caching: Updating cache when specific data changes in the database

How Django Signals Work

Django signals are based on the Python **signal** module, and they are used to notify certain senders when specific actions occur. They have three main components:

Signal: Represents a particular notification. Signals are created using the django.dispatch.Signal class. Each signal can have multiple "senders."

Sender: The component that sends a signal. In Django, senders are typically models, but they can be any Python object.

Receiver: A function or method that gets executed when a signal is sent. You can connect receivers to signals using the **@receiver** decorator or by calling **connect** on the signal object.

Examples of Step-Wise Creation

Example 1: User Registration Signal

in this example, we'll create a signal that sends a welcome email when a new user registers in a Django application.

Step 1: Create a Signal

```
# users/signals.py
from django.dispatch import Signal
user_registered = Signal()
```

Step 2: Define a Receiver Function

```
# users/receivers.py
from django.dispatch import receiver
from users.signals import user_registered
from django.core.mail import send_mail

@receiver(user_registered)
def send_welcome_email(sender, **kwargs):
    user = sender  # The new user that triggered the signal
    subject = 'Welcome to Our Website'
    message = f'Hello {user.username},\n\nThank you for registering on our website!'
    from_email = 'your@example.com'  # Replace with your email
    recipient_list = [user.email]

    send_mail(subject, message, from_email, recipient_list)
```

Step 3: Connect the Receiver Function

```
# users/receivers.py
from django.dispatch import receiver
from users.signals import user_registered
from django.core.mail import send_mail



@receiver(user_registered)
def send_welcome_email(sender, **kwargs):
    user = sender  # The new user that triggered the signal
    subject = 'Welcome to Our Website'
    message = f'Hello {user.username},\n\nThank you for registering on our website!'
    from_email = 'your@example.com'  # Replace with your email
    recipient_list = [user.email]

    send_mail(subject, message, from_email, recipient_list)
```

Step 4: Send the Signal

Send the signal in your user registration view.

```
# articles/views.py
from articles.models import Article
from articles.signals import pre_save_article

def create_article(request):
    if request.method == 'POST':
        form = ArticleForm(request.POST)
        if form.is_valid():
            new_article = form.save(commit=False)

            # Perform any additional processing here, e.g., setting the author or other fields

            new_article.save()
            pre_save_article.send(sender=new_article, instance=new_article)

            # Redirect to a success page or perform other actions
            return HttpResponseRedirect('/success/')
```

```
else:
    form = ArticleForm()

return render(request, 'create_article.html', {'form': form})
```

Class Based Views

Class Based Views:

Class-based views (CBVs) are a fundamental concept in web development, particularly in Python-based web frameworks like Django. They provide a way to structure and organize the logic of handling HTTP requests in a more object-oriented and reusable manner. Instead of defining views as functions, you define them as classes, which can offer a wide range of benefits, including code organization, code reuse, and readability.

Benefits of Class-Based Views:

Code Organization: CBVs encourage the organization of related view logic into classes. Each class typically represents a specific view or a group of related views. This makes the code more structured and easier to manage.

Code Reusability: CBVs promote the reuse of common functionality. You can create base view classes with generic methods and then inherit from them to create more specialized views. This reduces redundancy and makes your code more maintainable.

Readability: CBVs can improve the readability of your code, especially for complex views. The structure of class methods can make it clear what each view does, and you can use meaningful method names.

Mixins: CBVs often use mixins, which are small, reusable classes that can be combined to add specific behaviors to views. This is an excellent way to add functionality to your views without repeating code.

Built-in Generic Views: Many web frameworks, including Django, provide a set of generic class-based views for common tasks like creating, updating, and deleting objects. You can use these built-in views and customize them as needed.

In Django, Class-Based Views (CBVs) provide an alternative way to define views using Python classes rather than function-based views. They offer a more organized and reusable approach to handling HTTP requests in your Django application. CBVs come with various built-in views and mixins to make it easier to implement common patterns in web applications. Here are some of the main types of class-based views in Django, along with examples and explanations for each:

DetailView:

Description: DetailView is used for displaying a single object's details, typically retrieved from a model. It is commonly used for showing detailed information about a specific database record.

Example:

```
from django.views.generic import DetailView
from .models import MyModel

class MyModelDetailView(DetailView):
    model = MyModel
    template_name = 'mymodel_detail.html'
```

In this example, the DetailView is set to display the details of an instance of the MyModel model.

ListView:

Description: List View is used to display a list of objects from a model. It is often used to show a paginated list of items.

Example:

```
from django.views.generic import ListView
from .models import MyModel

class MyModelListView(ListView):
    model = MyModel
    template_name = 'mymodel_list.html'
    context_object_name = 'mymodels'
```

This List View displays a paginated list of objects from the MyModel model and assigns them to the context variable 'mymodels' for use in the template.

Create View:

Description: CreateView is used for handling the creation of new objects. It includes a form for user input and saving the object to the database.

Example:

```
from django.views.generic import CreateView
from .models import MyModel
```

```
class MyModelCreateView(CreateView):
    model = MyModel
    template_name = 'mymodel_form.html'
    fields = ['field1', 'field2']
```

This CreateView is configured to create instances of MyModel, using a form generated from the fields specified.

Update View:

Description: UpdateView is used for updating existing objects. It retrieves an object, displays a form pre-filled with its data, and updates it when the form is submitted.

Example:

python

Copy code

```
from django.views.generic import UpdateView
from .models import MyModel

class MyModelUpdateView(UpdateView):
    model = MyModel
    template_name = 'mymodel_form.html'
    fields = ['field1', 'field2']
```

This UpdateView allows users to edit and save an existing MyModel object.

DeleteView:

Description: DeleteView is used for deleting objects. It displays a confirmation page and deletes the object when confirmed.

Example:

```
from django.views.generic import DeleteView
from .models import MyModel
from django.urls import reverse_lazy
```

```
class MyModelDeleteView(DeleteView):
    model = MyModel
    template_name = 'mymodel_confirm_delete.html'
    success_url = reverse_lazy('mymodel-list')
```

This DeleteView provides a confirmation page for deleting a MyModel object and redirects to the 'mymodel-list' URL after deletion.