

PROJECT:09

“DJANGO”



SUBMITTED BY:
Manaswi M. Patil

INDEX

SR NO.	TOPICS	PG.NO.
1.	Introduction	3
2.	Django Create Project	6
3.	Django URLs	7
4.	Admin	14
5.	Django syntax	19
6.	PostgreySQL	26
7.	Add member	31
8.	Deploy django	32
9.	Elastic Beanstalk (EB)	33
	conclusion	36

1. INTRODUCTION

Django is widely used in the development of web applications. It is a high-level web framework for building web applications using the Python programming language. It is based on DRY (Don't Repeat Yourself), and comes with ready-to-use features like login system, database connection and CRUD operations (Create Read Update Delete) principles. Django and flask are the python frameworks. (Frameworks aim to simplify and streamline the development process by offering a foundation for building applications with common functionalities.)

Django follows the MVT design pattern (Model View Template).

- **Model** - The data you want to present, usually data from a database. The Model is responsible for handling data-related logic and represents the structure of the application's database.
- It defines the data models, including fields and their relationships, using Python classes.
- Django provides an Object-Relational Mapping (ORM) system, allowing developers to interact with the database using Python code rather than raw SQL.

```
from django.db import models

class MyModel(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
```

- **View** - A request handler that returns the relevant template and content - based on the request from the user. The View is responsible for processing user requests and returning appropriate responses.

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import MyModel

def my_view(request):
    data = MyModel.objects.all()
    return render(request, 'my_template.html', {'data': data})
```

- It contains the business logic of the application, handling user input, interacting with the Model, and returning data to be displayed.
- Views in Django are Python functions or classes that receive HTTP requests and return HTTP responses.
- **Template** - A text file (like an HTML file) containing the layout of the web page, with logic on how to display the data. The Template is responsible for defining the structure and layout of the HTML pages.
- It represents the presentation layer, where data from the View is rendered and displayed.

- Django uses its own template language, which is a mix of HTML and template tags, allowing dynamic content rendering.

```
<!DOCTYPE html>
<html>
<head>
  <title>My Template</title>
</head>
<body>
  <h1>My Data</h1>
  {% for item in data %}
    <p>{{ item.name }} - {{ item.description }}</p>
  {% endfor %}
</body>
</html>
```

Working of Django:

- Django receives the URL, checks the urls.py file, and calls the view that matches the URL.
- The view, located in views.py, checks for relevant models.
- The models are imported from the models.py file.
- The view then sends the data to a specified template in the template folder.
- The template contains HTML and Django tags, and with the data it returns finished HTML content back to the browser.

Installation steps:

- Install vs code editor in your system.
- check python install in your system or not → type this command in terminal or command prompt → python -version → if not install then you need to install latest version of python → <https://www.python.org/> → follow this link to install python.
- To install Django you must need 'pip' manager to install packages → 'pip -version'
- To create virtual environment → 'pip install virtualenv' in terminal
- Also installed Django version → type this command in terminal → 'python -m pip install django'
- To activate virtual environment → type this command → '.\venv\Scripts\activate'
- To check Django version → command like 'django-admin -version'
- Install flask → 'pip install flask'

Usage of Virtual environment:

Isolation of Dependencies: A virtual environment allows you to create a self-contained environment for your project with its own set of dependencies. This helps avoid conflicts between packages used in different projects.

Version Compatibility: Different projects may require different versions of Python or specific versions of third-party libraries. A virtual environment ensures that your project uses the correct versions, preventing compatibility issues.

Cleaner Project Structure: Including a virtual environment in your project keeps your project directory clean and organized. All the dependencies are isolated within the virtual environment folder, making it easier to manage and share your code.

Ease of Deployment: When deploying your Django project, you can share the virtual environment requirements (usually listed in a **requirements.txt** file) with others. This makes it simpler for collaborators or deployment tools to set up the same environment.

Ease of Testing: Virtual environments make it straightforward to test your application in different environments without affecting the system-wide Python installation. This is crucial for ensuring that your Django project works consistently across different setups.

Security: Isolating dependencies helps improve security. If a project requires a specific version of a library with known vulnerabilities, it won't impact other projects using different versions of that library.

➤ Python -m pip install Django → This command will give information about Django.

```
PS C:\WAWASWI\INTERNSHIP -PROJECT2023\python tutorial> python -m pip install Django
Collecting Django
  Obtaining dependency information for Django from https://files.pythonhosted.org/packages/50/1b/7536019fd20654919dcd81b475fee1e54f21bd71b2b4e094b2ab075478b2/Django-5.0.2-py3-none-any.whl.metadata
    Downloading Django-5.0.2-py3-none-any.whl.metadata (4.1 kB)
Collecting asgiref<4,>=3.7.0 (from Django)
  Obtaining dependency information for asgiref<4,>=3.7.0 from https://files.pythonhosted.org/packages/9b/80/b9051a4a07ad231558fcd8ffcc89232711b4e618c15cb7a392a17384bbe/asgiref-3.7.2-py3-none-any.whl.metadata
    Using cached asgiref-3.7.2-py3-none-any.whl.metadata (9.2 kB)
Collecting sqlparse<0.3.1 (from Django)
  Using cached sqlparse-0.4.4-py3-none-any.whl (41 kB)
Collecting tzdata (from Django)
  Obtaining dependency information for tzdata from https://files.pythonhosted.org/packages/65/58/f9c9e6be752e9fcb8b6a0ee9fb87e6e7a1f6bcab2cdc73f02bb7ba91ada0/tzdata-2024.1-py2.py3-none-any.whl.metadata
    Downloading tzdata-2024.1-py2.py3-none-any.whl.metadata (1.4 kB)
    Downloading Django-5.0.2-py3-none-any.whl (8.2 MB)
```

2. Django Create Project

- Create project with 'django-admin startproject Hello'
- Run this command to run Django project 'python manage.py runserver'

Django Create App

We will create an app that allows us to list and register members in a database

```
Hello > home > apps.py > ...
1  from django.apps import AppConfig
2
3
4  class HomeConfig(AppConfig):
5      default_auto_field = 'django.db.models.BigAutoField'
6      name = 'home'
7
```

Django Views:

- Django views are Python functions that takes http requests and returns http response, like HTML documents.
- A web page that uses Django is full of views with different tasks and missions. Views are usually put in a file called views.py located on your app's folder.
- views are responsible for processing user requests and returning appropriate responses. They handle the logic of your web application and determine what data is presented to the user.
- Views often render HTML templates to generate dynamic content. You can use the Django template engine to create templates with placeholders that are filled in with data when the view is called.
- Views interact with models to retrieve data from the database. You can query the database and pass the retrieved data to the template.
- Views process form submissions. They validate the form data, save it to the database, and redirect the user to another page or display error messages.

```
Hello > home > views.py > ...
1  from django.shortcuts import render,HttpResponse
2  from datetime import datetime
3  from home.models import Contact
4  # Create your views here.
5  def index(request):
6      context = {
7          "variable": "This is Demo1"
8      }
9      return render(request, 'index.html', context)
10     #return HttpResponse("This is Homepage")
11  def about(request):
12     return render(request, 'index.html')
13     #return HttpResponse("This is about page")
14  def services(request):
15     return render(request, 'services.html')
16     #return HttpResponse("This is services page")
17  def contact(request):
18     if request.method=="POST":
19         name=request.POST.get('name')
20         email=request.POST.get('email')
21         phone=request.POST.get('phone')
22         desc=request.POST.get('desc')
23         contact=Contact (name= name, phone= phone, email= email)
24         contact.save()
25     return render(request, 'contact.html')
26     #return HttpResponse("This is contact page")
27
```

3.Django URLs

- In your project's main urls.py file, you include the URLs of your individual apps using the include function.
- These patterns are essentially regular expressions that match the URLs of incoming requests.
- The term "contrib" stands for "contributed" or "contributory." The **django.contrib** module is a collection of optional, reusable components that are developed by the Django community and provided by the Django project itself. These components address common needs in web development and can be easily integrated into Django projects. The use of **contrib** modules brings additional functionality and features without the need to build everything from scratch.

```
Hello > Hello > urls.py > ...
1  """
2  URL configuration for Hello project.
3
4  The 'urlpatterns' list routes URLs to views. For more information please see:
5      https://docs.djangoproject.com/en/4.2/topics/http/urls/
6  Examples:
7  Function views
8      1. Add an import: from my_app import views
9      2. Add a URL to urlpatterns: path('', views.home, name='home')
10 Class-based views
11      1. Add an import: from other_app.views import Home
12      2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
13 Including another URLconf
14      1. Import the include() function: from django.urls import include, path
15      2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
16 """
17 from django.contrib import admin
18 from django.urls import path, include
19 admin.site.site_header = "Mini Toys World Admin"
20 admin.site.site_title = "Mini Toys World Portal"
21 admin.site.index_title = "Welcome to Mini Toys World "
22 urlpatterns = [
23     path('admin/', admin.site.urls),
24     path('', include('home.urls'))
25 ]
```

```
Hello > home > urls.py > ...
1  from django.contrib import admin
2  from django.urls import path
3  from home import views
4
5  urlpatterns = [
6      path("", views.index, name='home'),
7      path("about", views.about, name='about'),
8      path("services", views.services, name='services'),
9      path("contact", views.contact, name='contact'),
10
11
12  ]
```

Click this link to go ahead:

```
PS C:\MANASWI\DJANGO\django in one video\hello> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
February 13, 2024 - 22:52:34
Django version 5.0.1, using settings 'Hello.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

This is contact page.

🔒 127.0.0.1:8000/contact

Contact us

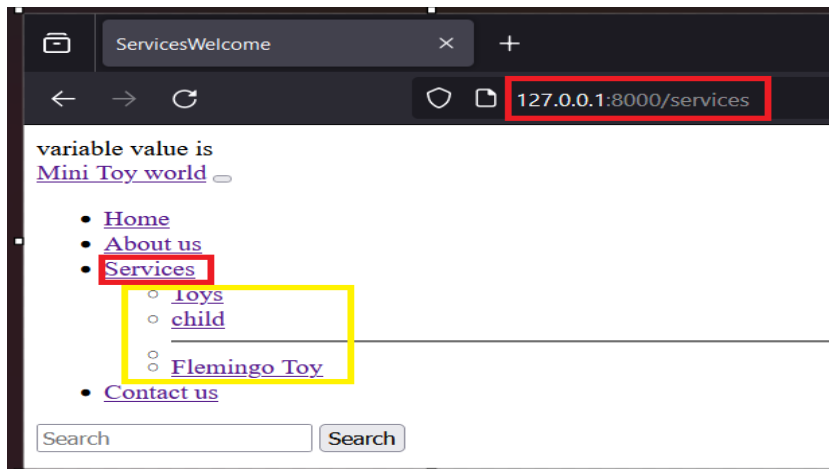
Name

Email Address

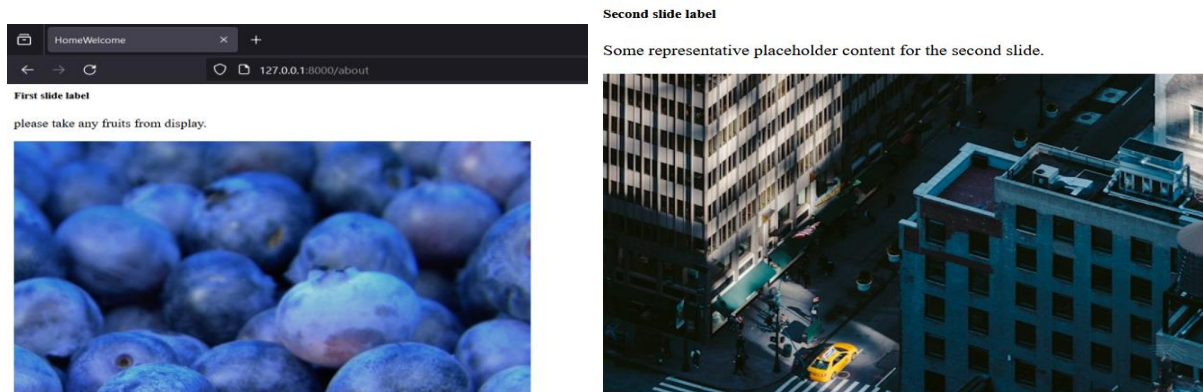
Phone Number

Tell me about what you want to contact

This is services page.



This is about page.

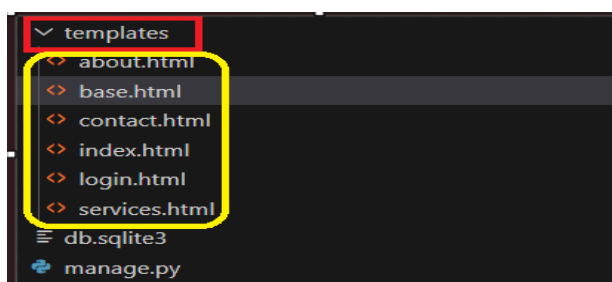


Third slide label

Some representative placeholder content for the third slide.

Django Templates:

- Create a 'templates' folder outside the 'hello' folder, and create a HTML file named 'index.html'. same as this Create this all files of html.




- Django templates are files that define how dynamic content should be displayed in a web application. Django uses its own template language, which is a lightweight and expressive

markup language, to create dynamic HTML or other output formats. Template files typically have the extension **.html**, but Django supports other template engines as well.

1. **Syntax:** Django template syntax uses double curly braces `{{ ... }}` for variables and template tags enclosed in curly braces and percent signs `{% ... %}` for control flow and other logic.
2. **Variables:** Variables inside `{{ ... }}` are placeholders for dynamic data. In the context of Django, these variables are often replaced with actual values when the template is rendered.
3. **Template Tags:** Template tags inside `{% ... %}` provide logic and control flow in templates. They include for loops, if statements, and other constructs.
4. **Filters:** Filters modify the output of variables in the template. They are applied using the pipe `|` symbol.
5. **Template Inheritance:** Django supports template inheritance, allowing you to create a base template and extend or override specific blocks in child templates.
6. **Static Files:** Template tags are used to include static files like CSS and JavaScript into templates.
7. **URLs:** Template tags are used to generate URLs dynamically.
8. **Forms:** Django templates are commonly used for rendering HTML forms and handling form submissions.
9. **Conditionals:** Template tags support conditionals for rendering content based on certain conditions.

Django Models:

A model is a Python class that defines the structure of a database table and the fields it contains. Django models provide an abstraction layer for interacting with databases, allowing developers to work with high-level Python code rather than SQL queries. Models are used to define the data structure of the application and handle database operations such as querying, creating, updating, and deleting records.

```
Hello > home >  models.py > ...
1  from django.db import models
2  # Create your models here.
3  class Contact(models.Model):
4      name =models.CharField(max_length=122)
5      email =models.CharField(max_length=122)
6      phone =models.CharField(max_length=12)
7      desc =models.TextField
8      date= models.DateField
9
```

- A model class is a Python class that inherits from `'django.db.models.Model'`. Each attribute of the class represents a field in the corresponding database table.
- Fields in a model define the type of data that can be stored in the corresponding database column. Django provides various field types, such as CharField, IntegerField, DateField, ForeignKey, and more.
- Django automatically adds an integer field named `id` as the primary key for every model.
- **Model Instances:** Model instances represent individual records in the database. You can create, retrieve, update, and delete records using model instances.

- **Database Tables:** Django automatically generates database tables based on the model classes defined in your application. You can use the migrate management command to apply database schema changes.
- **QuerySets:** QuerySets are used to query the database and retrieve records based on certain conditions. They allow for filtering, ordering, and aggregating data.
- **Model Forms:** Django models are often used in conjunction with forms to simplify the process of creating and updating records through web forms.

Django Insert Data:

You can insert (or create) data into the database using models and model instances.

1. **Define Model:** First, define your model in the **models.py** file of your Django app. This involves creating a class that inherits from **django.db.models.Model** and adding fields to represent the columns in your database table.
2. **Create Migration:** After defining your model, you need to create a migration to apply the changes to the database. Commands are:
 - `python manage.py makemigrations`
 - `python manage.py migrate`

Django Update Data

In Django, updating data involves retrieving existing records from the database, modifying their attributes, and saving the changes back to the database

Retrieve the Record: Use a query to retrieve the record you want to update. You can use the **get()** method to retrieve a single record based on certain conditions.

1. **Modify the Attributes:** Once you have the record, modify its attributes as needed.
2. **Save the Changes:** Call the **save()** method on the record to persist the changes to the database.
3. Alternatively, you can perform the update in a single step using the **update()** method: The **update()** method doesn't call the **save()** method on each record individually, making it more efficient for bulk updates.
4. **Update Multiple Records:** If you want to update multiple records based on certain conditions, you can use the **filter()** method to retrieve a queryset and then call **update()** on the queryset.
5. **Using Forms:** If you are updating data through a form, you can use a model form similar to the creation process. Retrieve the record, populate the form with the existing data, validate the form data, and save the form to update the record.

```
Hello > home > views.py contact
12 def about(request):
13     return render(request, 'index.html')
14     #return HttpResponse("This is about page")
15 def services(request):
16     return render(request, 'services.html')
17     #return HttpResponse("This is services page")
18 def contact(request):
19     if request.method=="POST":
20         name=request.POST.get('name')
21         email=request.POST.get('email')
22         phone=request.POST.get('phone')
23         desc=request.POST.get('desc')
24         contact=Contact (name= name, phone= phone, email= email)
25         contact.save()
26     messages.success(request, "message has been sent")
27     return render(request, 'contact.html')
28     #return HttpResponse("This is contact page")
29
```

🚀 Django Delete Data:

1. If you have a model instance, you can use the `delete()` method to remove the corresponding record from the database.
2. **Delete Record by Query:** You can use the `filter()` method to retrieve a queryset of records based on specific conditions and then call `delete()` on the queryset to remove the matching records from the database.
3. **Delete All Records:** To delete all records in a table, you can use the `all()` method to retrieve all records and then call `delete()`.
4. **Using Forms:** If you are deleting data through a form or user interface, you can create a view that handles the deletion process. Retrieve the record based on user input or form submission and call the `delete()` method.
5. **Soft Deletion:** If you want to implement a soft delete (marking records as deleted without physically removing them from the database), you can add a boolean field, such as `is_deleted`, to your model and update its value instead of using `delete()`.

🚀 Django Update Model:

1. **Retrieve the Record:** Use a query to retrieve the record you want to update. For example, if you want to update a record with the id equal to 1:
2. **Modify the Attributes:** Once you have the record, modify its attributes as needed.
3. **Save the Changes:** Call the `save()` method on the record to persist the changes to the database.
4. You can perform the update in a single step using the **update() method**. The `update()` method is more efficient for bulk updates as it doesn't call the `save()` method on each record individually.
5. **Update Multiple Records:** If you want to update multiple records based on certain conditions, you can use the `filter()` method to retrieve a queryset and then call `update()` on the queryset.

6. **Using Forms:** If you are updating data through a form, you can use a model form similar to the creation process. Retrieve the record, populate the form with the existing data, validate the form data, and save the form to update the record.

Display Data:

To display data in Django, you typically use views and templates.

1. **Retrieve Data in Views:** In your views (defined in `views.py`), retrieve the data from your models using Django's ORM.
2. **Create a Template:** Create an HTML template file (e.g., `display_data.html`) where you want to render the data.
3. **Configure URL Patterns:** Configure the URL patterns in your `urls.py` to map the view to a URL.
4. **Access the Data:** Start the development server (`python manage.py runserver`) and navigate to the this url '`http://127.0.0.1:8000`' to view the data. Your Django app should now display the data from your model in the specified template .

Add Main Index Page:

1. **Create a New Template for Index Page:** Create a new file named `index.html` in the templates directory. In this template, we extend the master template (`base.html`), set the page title, and provide some introductory content.
2. **Update the Master Template (`base.html`):** Update the master template to include a link to the new index page in the navigation. We added a new link to the index page (`{% url 'index' %}`) in the navigation.
3. **Update Views and URL Patterns:** Update the `views.py` file to include a view for the index page→ Update the URL patterns (`urls.py`) to include a pattern for the index view:
4. **Test Your Application:** Start the development server→`python manage.py runserver`

Django 404 Template:

In Django, you can create a custom 404 error page to be displayed when a page is not found.


1. **Create a 404 Template:** Create a new HTML file named `404.html` in your templates directory. This file will serve as the custom 404 error page. Customize the content and styling of the `404.html` file according to your needs.
2. **Update the `urls.py` File:** In your Django project's `urls.py` file, you can include a handler for 404 errors and specify the custom template.

3. **Create a Custom 404 View:** In your `views.py` file, create a view function that will handle 404 errors. You can customize this function to render the custom 404 template. Make sure to import this view function in your `urls.py` file.
4. **Test the 404 Page:** Start the development server → `python manage.py runserver` .

Add Test View:

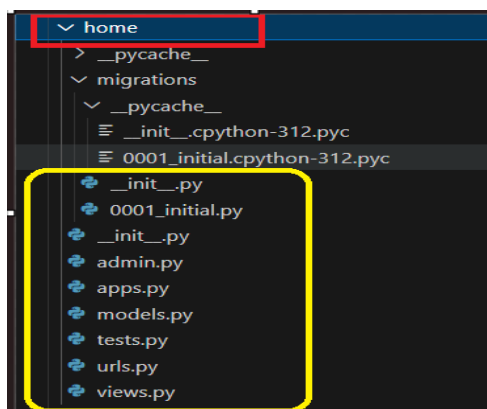
1. **Create a New View:** In your `views.py` file, create a new view function. This view function simply renders a template named `test_view.html` and passes a message to it.
2. **Create a Template for the Test View:** Create a new HTML file named `test_view.html` in the `templates/hello` directory. This template displays the message passed from the view and includes a link to return to the home page.
3. **Update URL Patterns:** Update your `urls.py` file to include a URL pattern for the new test view.
4. **Test Your Application:** Start the development server → `python manage.py runserver` .

4.Admin

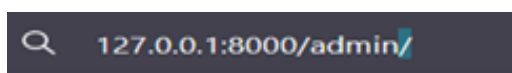
 **Create User:** To create a user in a Django application, you can use the Django's built-in `createsuperuser` command for the admin site or create users programmatically in your views or management commands.

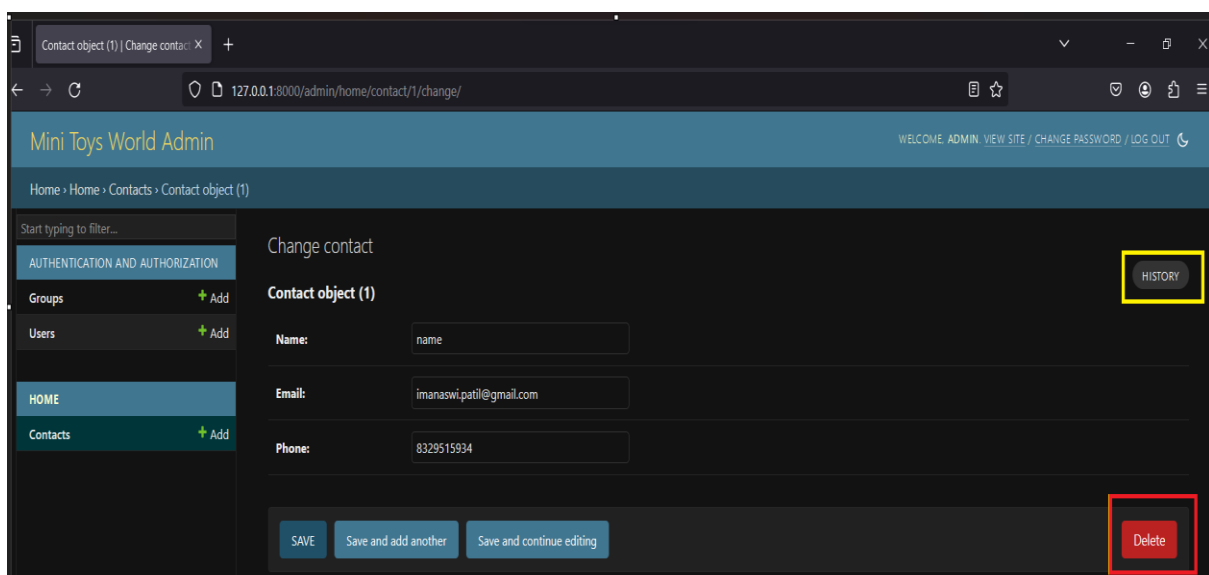
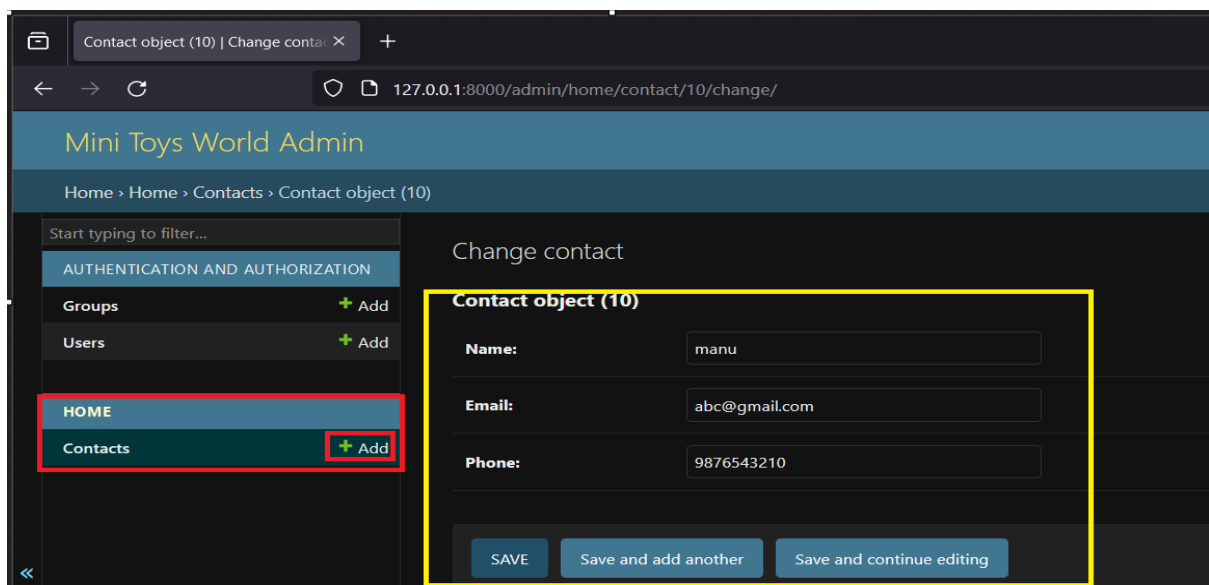
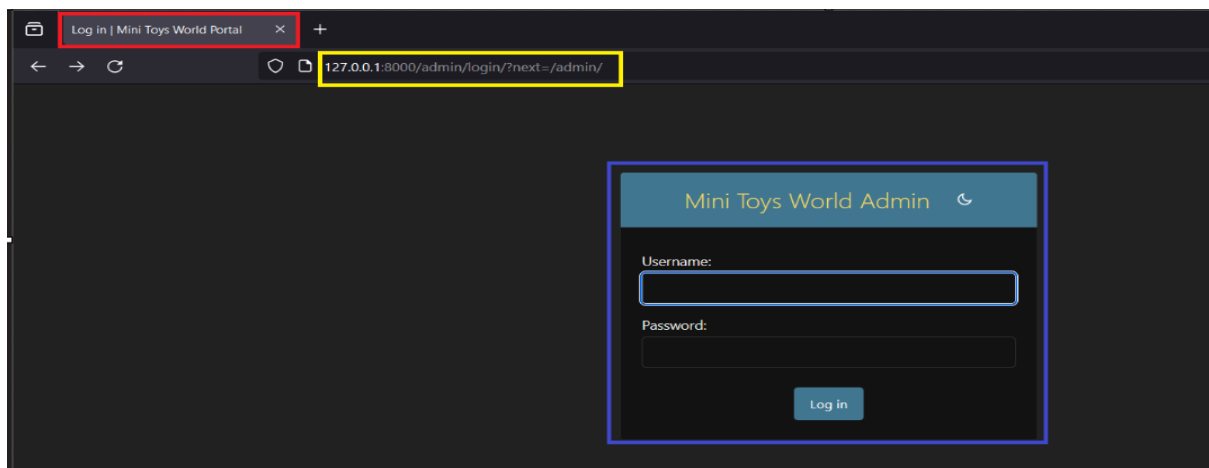
- Using `createsuperuser` Command → Open a terminal in your project directory → Run the following command → `python manage.py createsuperuser` → Follow the prompts to enter a username, email address, and password for the superuser.
- If you want to create users programmatically in your views, management commands, or other parts of your Django application, you can use the User model from `django.contrib.auth.models`. If you want to include models in your Django project, you need to define them in the `'models.py'` file of your app.
- Create a Superuser: Django's admin interface requires a superuser account. Open a terminal and run the following command in your project directory → Follow the prompts to create a superuser account by providing a username, email, and password.
- Configure Admin Site: In your app's `admin.py` file (create one if it doesn't exist), register the models you want to manage through the admin interface. For example:
- Include Admin URLs in Project URLs: In your project's `urls.py` file, include the admin URLs by importing the admin module and using the `admin.site.urls` pattern. For example:
- Run the Development Server: Start the development server.

In home folder will create this all files with extension `'py'`.



- To open login page → click on url (return in terminal) → open it in browser → add `admin/` to open this page. You can add multiple contacts using add button.



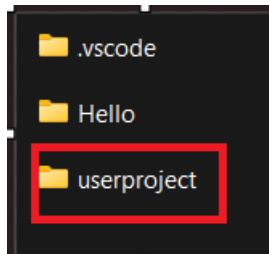


Django Admin:

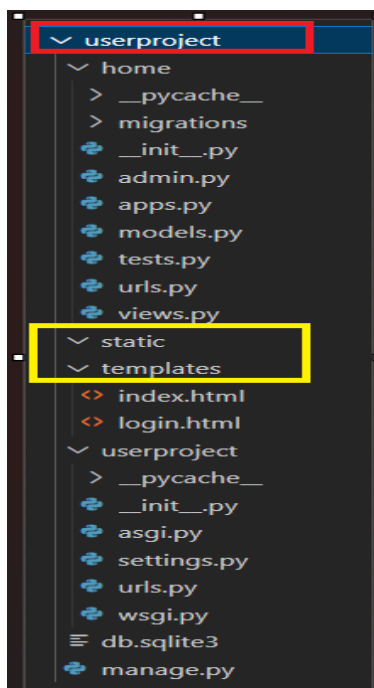
Django Admin is a powerful and built-in administration interface that allows you to manage your application's data, including models, users, groups, and more.

🚦 Login page in Django:

Open 'files' from window → click to collapse 'Django in one video' folder → right click on window → open new terminal → type command as 'Django-admin startproject userproject' → folder created → open with vs code.



- Follow this command → `python manage.py makemigrations` → `python manage.py migrate` → `python manage.py startapp home` → open 'setting.py' of userproject. --> create static and template folder → in templates folder create → `index.html` and make changes → `login.html` and make changes → open `urls.py` of useproject and add some content → now make `urls.py` file in home and make code into it.
- Now we make changes in this folders(userproject).



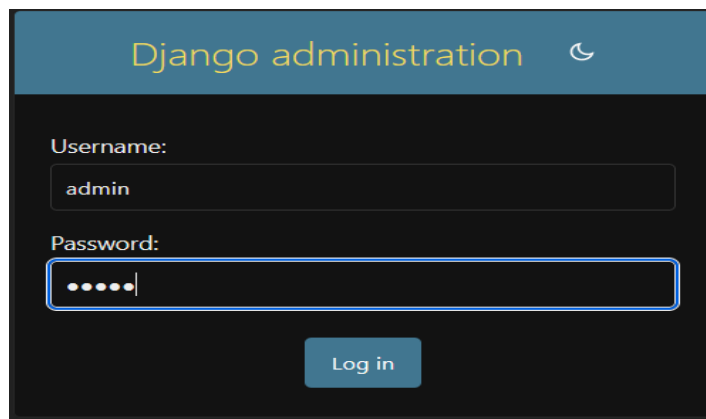
🚦 **Creating a Superuser:** Before using the admin interface, you need to create a superuser account. Run the following command in the terminal and follow the prompts → `python manage.py createsuperuser` .


```

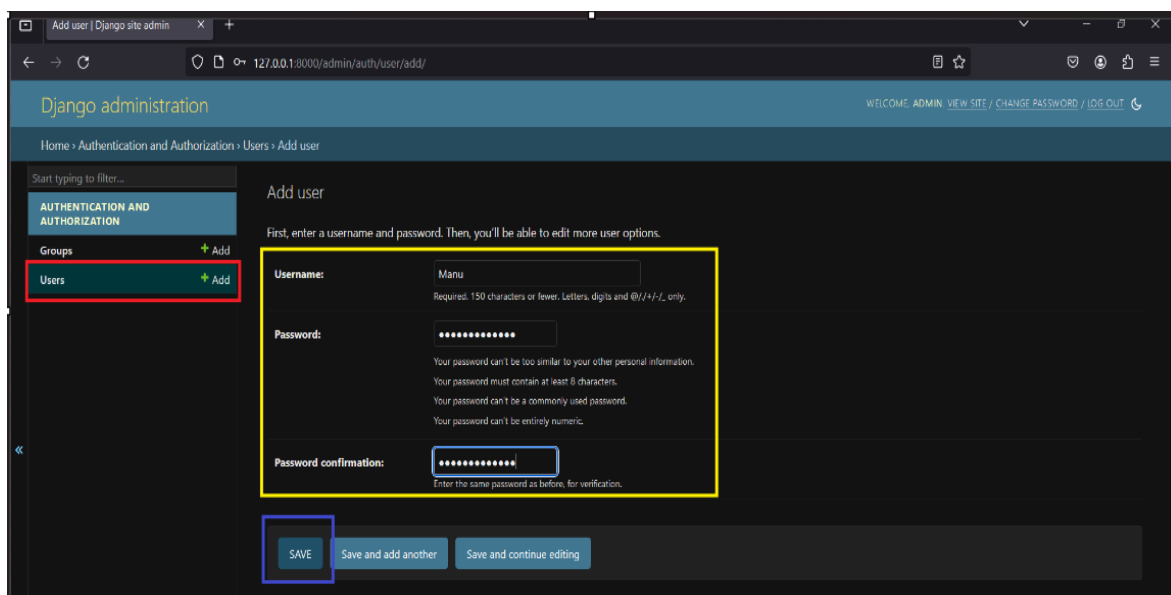
PS C:\MANASWI\DJANGO\django in one video> cd userproject
PS C:\MANASWI\DJANGO\django in one video\userproject> python manage.py createsuperuser
Username (leave blank to use 'hp'): admin
Email address:
Password:
Password (again):
Error: Blank passwords aren't allowed.
Password:
Password (again):
Error: Blank passwords aren't allowed.
Password:
Password (again):
Error: Blank passwords aren't allowed.
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
PS C:\MANASWI\DJANGO\django in one video\userproject> █

```

127.0.0.1:8000/admin/



The image shows the Django administration login interface. At the top, there's a header with 'Django administration' and a moon icon. Below the header, there are two input fields: 'Username:' with the value 'admin' and 'Password:' with masked characters. A 'Log in' button is positioned at the bottom right of the form area.



The image shows the 'Add user' page in the Django administration interface. The left sidebar has a red box around the 'Users' link under 'AUTHENTICATION AND AUTHORIZATION'. The main content area has a yellow box around the 'Add user' form. The form includes fields for 'Username:' (with the value 'Manu'), 'Password:', and 'Password confirmation:'. Below the form, there are three buttons: 'SAVE' (highlighted with a blue box), 'Save and add another', and 'Save and continue editing'.

Manu | Change user | Django site

127.0.0.1:8000/admin/auth/user/2/change/

Django administration

WELCOME, ADMIN | [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Authentication and Authorization > Users > Manu

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#)

Users [+ Add](#)

Change user

Manu [HISTORY](#)

Username:
Required. 150 characters or fewer. Letters, digits and @/+/./_ only.

Password: **algorithm:** pbkdf2_sha256 **iterations:** 720000 **salt:** jlyeaD***** **hash:** BfseBE*****
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Personal info

First name:

Last name:

Email address:

Important dates

Last login:

Date: [Today](#) | [📅](#)

Time: [Now](#) | [🕒](#)

Note: You are 5.5 hours ahead of server time.

Date joined:

Date: [Today](#) | [📅](#)

Time: [Now](#) | [🕒](#)

Note: You are 5.5 hours ahead of server time.

[SAVE](#) [Save and add another](#) [Save and continue editing](#) [Delete](#)

[🔍](#) The user "Manu" was changed successfully.

Select user to change [ADD USER +](#)

[Search](#)

Action: [Go](#) 0 of 2 selected

<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	Manu	imanaswi.patilmanu5@gmail.com	Manu	Patil	<input type="radio"/>
<input type="checkbox"/>	admin				<input checked="" type="radio"/>

2 users

FILTER

[👁 Show counts](#)

[↓ By staff status](#)

All
Yes
No

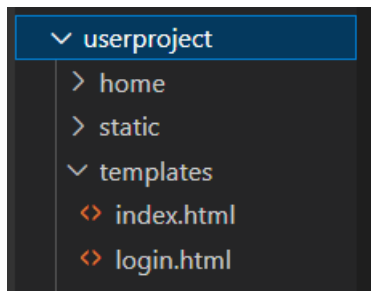
[↓ By superuser status](#)

All
Yes
No

[↓ By active](#)

All
Yes
No

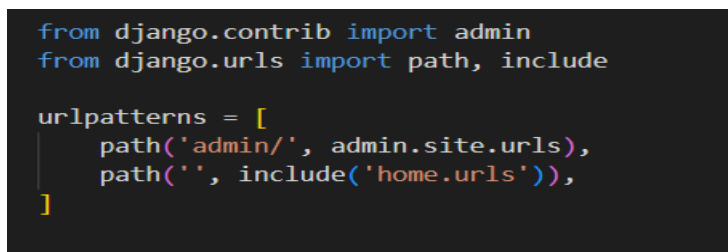
- Make changes in user project templates files(index.html and login.html).



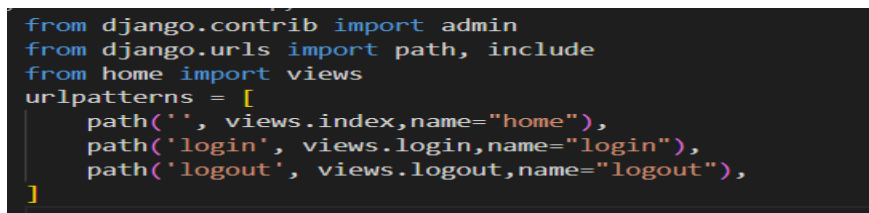
- In 'settings.py' file of userproject make some changes or add content.



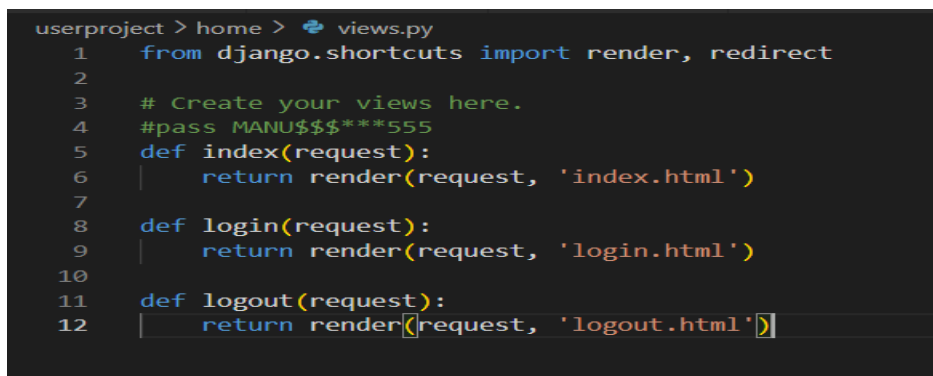
- Click userproject folder → click urls.py file → make changes.



- Now create urls.py file in home app which you created.



- Make connection with views.py from userproject.



- Copy the class name in apps.py file.

```
from django.apps import AppConfig

class HomeConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'home'
```

Please sign in

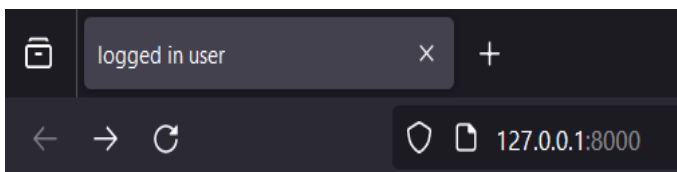
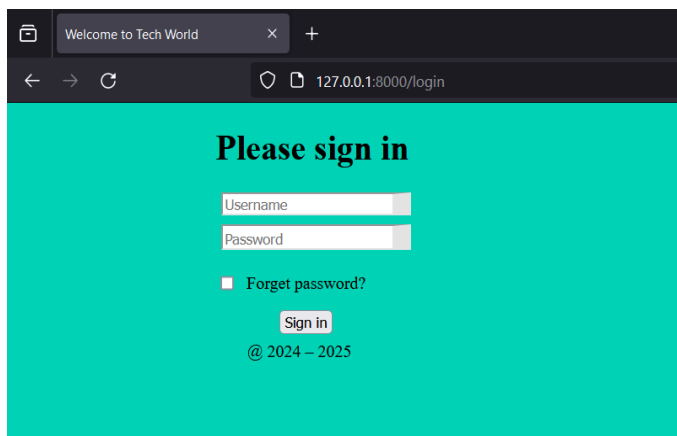
Enter your email

Enter Password

☐ Remember me

Sign in

✚ After css to page



welcomeadmin

Hey this is the tutorial for beginners.

[Logout](#)

Csrf token{% csrf_token %} :

A CSRF (Cross-Site Request Forgery) token is a security measure implemented in web applications to protect against CSRF attacks. CSRF is a type of attack where an attacker tricks a user's browser into making an unintended and potentially malicious request on a web application where the user is authenticated. The CSRF token is a unique token generated by the server and included in the web page or sent to the client in some way. The token is then included in each subsequent request made by the user, typically in the form of a hidden field in a form. When the user submits a form or performs an action that requires authentication, the server checks the CSRF token to ensure that the request is legitimate and not a result of a CSRF attack.

5.Django Syntax

Django If Else:

In Django templates, you can use the ' {% if %} {% else %} {% endif %}' template tag to create conditional statements.

Django For Loop:

In Django templates, you can use the ' {% for %} {% endfor %}' template tag to iterate over lists, querysets, and other iterable objects. Here's a basic example:

Looping Through a List:

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

Looping Through a QuerySet:

```
<table> <tr> <th>Name</th> <th>Age</th> </tr> {% for person in people_queryset %} <tr>
<td>{{ person.name }}</td> <td>{{ person.age }}</td> </tr> {% endfor %} </table>
```

Looping Through a Range of Numbers:

```
<ul> {% for i in 1|length|add:"-1" %} <li>{{ i }}</li> {% endfor %} </ul>
```

Nested For Loops:

```
<table> {% for row in matrix %} <tr> {% for cell in row %} <td>{{ cell }}</td> {% endfor
%} </tr> {% endfor %} </table>
```

Loop Variables: Inside the {% for %} loop, you can access loop-related variables.

- forloop.counter: The current iteration (1-indexed).
- forloop.counter0: The current iteration (0-indexed).
- forloop.revcounter: The reverse iteration (1-indexed).
- forloop.revcounter0: The reverse iteration (0-indexed).
- forloop.first: True if this is the first iteration.
- forloop.last: True if this is the last iteration.

QuerySets:

- Queryset is a representation of a database query that you can construct, filter, and execute. It allows you to interact with your database using a high-level, Pythonic interface. Django's models provide a convenient way to query your database and retrieve data.
- In Django, a QuerySet is a representation of a database query and is used to interact with the database. It allows you to retrieve, filter, and manipulate data from the

database. Below is an introduction to QuerySets, along with examples of the `get()`, `filter()`, and `order_by()` methods.

- Open `views.py` → Run this command in new terminal → 'python manage.py shell'.

```
>>> from home.models import Contact
>>> Contact.objects.all()
<QuerySet [ <Contact: Contact object (1)>, <Contact: Contact object (2)>, <Contact: Contact object (3)>, <Contact: Contact object (4)>, <Contact: Contact object (5)>, <Contact: Contact object (6)>, <Contact: Contact object (7)>, <Contact: Contact object (8)>, <Contact: Contact object (10)>, <Contact: Contact object (11)>, <Contact: Contact object (12)>, <Contact: Contact object (13)>, <Contact: Contact object (14)>, <Contact: Contact object (15)>, <Contact: Contact object (16)>, <Contact: Contact object (17)>, <Contact: Contact object (18)>, <Contact: Contact object (19)>, <Contact: Contact object (20)> ]>
```

This is used for filtering queries or searching same from queryset.

```
>>> Contact.objects.all()[0]
<Contact: Contact object (1)>
>>> Contact.objects.all()[0]
<Contact: Contact object (1)>
>>> Contact.objects.all()[0].name
'name'
>>> Contact.objects.all()[0].email
'imanaswi.patil@gmail.com'
>>> █
```

Filtering:

We can find any queryset using this command

```
>>> Contact.objects.filter(email="email")
<QuerySet []>
>>> Contact.objects.filter(name="name")
<QuerySet [ <Contact: Contact object (1)> ]>
>>> █
```

To find last or first query from queryset used `last()`, `first()`

```
>>> Contact.objects.all().last()
<Contact: Contact object (20)>
>>> Contact.objects.all().first()
<Contact: Contact object (1)>
>>> █
```

1. QuerySet Introduction:

A QuerySet is a collection of database queries that can be used to filter and retrieve data from your database. It is obtained by querying a Django model. Here's a basic example:

2. QuerySet `get()` Method:

The `get()` method retrieves a single object that matches the specified conditions. If multiple objects are found or none are found, it raises an exception. Example:

3. QuerySet `filter()` Method:

The `filter()` method is used to retrieve a set of objects that match the specified conditions. It returns a `QuerySet` containing all the objects that meet the criteria. Example:

4. `QuerySet order_by()` Method:

The `order_by()` method is used to sort the objects in the `QuerySet` based on one or more fields. You can specify the order (ascending or descending) for each field.

Static Files :

- Static files refer to files that remain unchanged during the execution of a program or a website. These files typically include things like images, stylesheets, scripts, and other resources that are not dynamically generated. Serving static files efficiently is a common requirement for web applications.
- - Static files refer to files that remain unchanged during the execution of a program or a website. These files typically include things like images, stylesheets, scripts, and other resources that are not dynamically generated. Serving static files efficiently is a common requirement for web applications.
- Images: JPEG, PNG, GIF, SVG files.
- Stylesheets: CSS files.
- Scripts: JavaScript files.
- Fonts: Font files like TTF, OTF, WOFF.

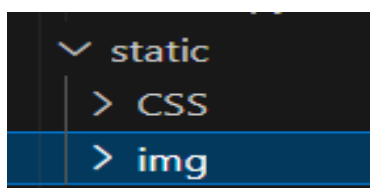
django (Python web framework): Django has a built-in `django.contrib.staticfiles` app that helps manage static files during development and allows you to collect them in a single location for deployment.

Express (Node.js framework): In Express, you can use the `express.static` middleware to serve static files.

Flask (Python web framework): Flask allows you to serve static files using the `static` folder in your project.

Add Static Files:

1. Django (Python)→In a Django project→Create a folder named `static` in your app directory→Place your static files (CSS, JavaScript, images, etc.) inside this folder. →To use these static files in your templates, load the `static` tag.



2. Express (Node.js)→In an Express project→Create a public folder in your project→Place your static files inside this folder.

3. Flask (Python)→In a Flask project→Create a static folder in your project→Place your static files inside this folder.

Install White Noise:

WhiteNoise is a Python package that allows you to serve static files efficiently in a Django web application. It can be used to simplify the process of serving static files during development and in production.:

- Activate your virtual environment (if you're using one)→source /path/to/your/venv/bin/activate →Install WhiteNoise using pip→pip install whitenoise→Configure WhiteNoise in your Django project→In your settings.py file, add whitenoise.middleware.WhiteNoiseMiddleware to the MIDDLEWARE list:
- Configure static files settings→Collect static files→If you haven't already, run the following command to collect your static files→python manage.py collectstatic →This command collects static files from your apps and places them in the designated directory (as specified by STATIC_ROOT in your settings).
- Run your Django application→python manage.py runserver →WhiteNoise will now serve your static files during development.

Collect Static Files:

Collecting static files in a Django project involves gathering all the static files from your apps and placing them in a single directory, ready to be served by your web server or WhiteNoise in production. Open a terminal or command prompt.

- Navigate to your Django project directory, where the manage.py file is located→Run the following command→python manage.py collectstatic →If you are using a virtual environment, make sure it is activated.

Add Global Static Files:

In a Django project, you can add global static files that are not associated with any specific app. These static files can include common stylesheets, scripts, or images that are used throughout your entire project.

Create a static folder in the root of your Django project (where the manage.py file is located).

1. Inside the static folder, create subdirectories for different types of static files (e.g., css, js) and place your global static files inside those subdirectories.

2. In your settings.py file, configure the STATICFILES_DIRS setting to include the path to the global static files folder. Add the following line → `STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static'),]` → in settings.py file.
3. In your templates, you can now reference these global static files using the `{% load static %}` and `{% static %}` template tags.

Add Styles to the Project:

To add styles to your Django project, you can create or modify CSS files and link them in your HTML templates

Global Styles:

1. Create a static/css folder in the root of your Django project (next to the manage.py file).
2. Add your global styles in global.css. For example:
3. In your base template or any template where you want to include these global styles, load the static tag and link to the global CSS file:

App-Specific Styles:

1. In each Django app that needs its styles, create a static/css folder.
2. Add styles to the respective app-specific CSS files (e.g., app1_styles.css and app2_styles.css).
3. In the HTML templates of each app, load the static tag and link to the app-specific CSS files. → Repeat this process for each app that requires its styles. When you run `python manage.py collectstatic`, Django will collect these static files into a central location for serving.

5.PostgreSQL

PostgreSQL, often referred to as "Postgres," is an open-source relational database management system (RDBMS). It is known for its robustness, extensibility, and adherence to SQL standards. PostgreSQL is designed to handle various types of workloads, from small single-machine applications to large and complex enterprise systems.

1. **Open Source:** PostgreSQL is released under the PostgreSQL License, which is a permissive open-source license. This means that you can use, modify, and distribute PostgreSQL freely.
2. **Relational Database:** PostgreSQL is a relational database system, which means it organizes data into tables with rows and columns. It supports the SQL (Structured Query Language) for defining and manipulating the data.
3. **ACID Compliance:** PostgreSQL follows the principles of ACID (Atomicity, Consistency, Isolation, Durability), ensuring that transactions are reliable, secure, and maintain data integrity.
4. **Extensibility:** One of the notable features of PostgreSQL is its extensibility. Users can define their data types, operators, functions, and even programming languages, allowing for customizations and support for specialized use cases.
5. **Data Types:** PostgreSQL provides a wide range of built-in data types, including integer, text, date, JSON, arrays, and more. Additionally, users can create custom data types to meet specific requirements.
6. **Concurrency Control:** PostgreSQL handles multiple users accessing the database simultaneously through sophisticated concurrency control mechanisms, ensuring data consistency.
7. **Scalability:** PostgreSQL can scale horizontally (across multiple servers) or vertically (by adding more resources to a single server) to accommodate growing workloads.
8. **Support for Geospatial Data:** PostgreSQL has excellent support for geospatial data types and queries, making it a popular choice for applications involving geographic information systems (GIS).
9. **Active Community:** PostgreSQL has a large and active community of developers, contributors, and users. This community-driven development model ensures continuous improvement and support.

10. **Triggers and Stored Procedures:** PostgreSQL supports triggers and stored procedures, allowing developers to define custom actions that automatically respond to certain events or conditions.

PostgreSQL Intro:

PostgreSQL, often referred to as "Postgres," is a powerful and feature-rich open-source relational database management system (RDBMS). Developed over decades, PostgreSQL has gained a reputation for its reliability, extensibility, and adherence to SQL standards. Open Source: PostgreSQL is released under the PostgreSQL License, a permissive open-source license. This allows users to freely use, modify, and distribute the software.

1. **Relational Database Management System (RDBMS):** PostgreSQL follows a relational model, organizing data into tables with rows and columns. It uses SQL (Structured Query Language) for defining, querying, and manipulating data.
2. **ACID Compliance:** PostgreSQL adheres to ACID principles (Atomicity, Consistency, Isolation, Durability), ensuring the reliability and integrity of transactions even in the event of system failures.
3. **Extensibility:** PostgreSQL is highly extensible, allowing users to define custom data types, operators, functions, and aggregates. This flexibility makes it suitable for diverse and specialized use cases.
4. **Data Types:** The RDBMS supports a broad range of built-in data types, including integers, text, dates, arrays, JSON, and more. Users can also create their custom data types to address specific requirements.
5. **Concurrency Control:** PostgreSQL employs advanced concurrency control mechanisms to manage multiple users accessing the database concurrently, ensuring consistency and avoiding conflicts.
6. **Scalability:** PostgreSQL can scale both vertically (by adding more resources to a single server) and horizontally (by distributing data across multiple servers) to handle increased workloads.
7. **Advanced Features:** It offers various advanced features such as support for full-text search, geospatial data, JSON and XML data types, and powerful indexing capabilities.
8. **Community and Support:** PostgreSQL has a large and active community of developers and users. This community-driven development model ensures constant improvement, timely bug fixes, and a wealth of resources for users.

9. Triggers and Stored Procedures: PostgreSQL supports triggers, which are actions that automatically respond to specific events, and stored procedures, allowing developers to define custom functions to be executed within the database.

Create AWS Account:

Creating an AWS account in a Django application typically involves integrating AWS services for cloud-related functionalities. Here's a general guide on how you might handle this:

1. AWS SDK for Python (Boto3): First, you'll need to install the AWS SDK for Python, commonly known as Boto3. You can do this using pip → `pip install boto3`
2. AWS Account Setup: Before you can use AWS services programmatically, you'll need to set up an AWS account, as mentioned in the previous response.
3. AWS Access Key and Secret Key: In order to interact with AWS services programmatically, you need access to AWS credentials (Access Key ID and Secret Access Key). Create an IAM (Identity and Access Management) user in the AWS Management Console, assign the necessary permissions, and obtain the access key and secret key.
4. Configure Boto3: In your Django project, configure Boto3 with your AWS credentials. You can do this in your Django settings.
5. Using AWS Services in Django: Now you can use Boto3 to interact with AWS services directly from your Django application. For example, using Amazon S3 for storing files:
6. Security Considerations: Never hardcode your AWS credentials directly in your code. Use environment variables, AWS credentials file, or IAM roles.

Create Database in RDS:

Creating a database in Amazon RDS (Relational Database Service) for use with a Django application involves a few steps. Below is a guide on how to create a PostgreSQL database in RDS for a Django project.

Prerequisites:

1. AWS Account: Ensure that you have an AWS account. If you don't have one, follow the steps mentioned earlier to create an AWS account.
2. Django Project: Have a Django project ready. If you don't have a Django project yet, you can create one using the following command → `django-admin startproject your_project`.

Steps to Create a PostgreSQL Database in RDS:

1. Go to AWS Management Console → Log in to the [AWS Management Console](#) → Navigate to the RDS service.

2. Create a New RDS Instance→Click on the "Create database" button→Choose the "Standard create" option.
3. Select Database Engine and Version:Choose the PostgreSQL database engine→Select the PostgreSQL version you want to use.
4. Specify DB Details:Set the DB instance identifier (a unique name for your database)→Set the Master username and password.
5. Configure DB Instance:Choose the instance type, storage type, and allocated storage according to your requirements→Configure additional settings like VPC, Subnet group, and publicly accessible options.
6. Database Authentication:Set the authentication method. You can choose password authentication.
7. Configure Advanced Settings:You can configure additional settings like database name, port, and parameter groups.
8. Add Tags (Optional):You can add tags for easier resource management.
9. Review and Launch:Review your configuration settings→Click on "Create database."
10. Wait for Database Creation:It may take several minutes for the database to be created. You can monitor the progress on the RDS dashboard.
11. Connect to Database:Once the database is ready, you can find the endpoint on the RDS dashboard→Use this endpoint, along with the master username and password, to connect to the database from your Django application.
12. Apply Migrations:Run the following Django management command to apply migrations→python manage.py migrate

Connect to Database:

Connecting Django to a PostgreSQL database involves configuring your Django project's settings to use the appropriate database engine, name, user credentials, and other settings.

1. Install psycopg2: psycopg2 is the PostgreSQL adapter for Python that Django uses to connect to PostgreSQL databases. Install it using→'pip install psycopg2 '.

```
PS C:\MANASWI\DJANGO\django in one video> pip install psycopg2
Collecting psycopg2
  Downloading psycopg2-2.9.9-cp312-cp312-win_amd64.whl.metadata (4.5 kB)
  Downloading psycopg2-2.9.9-cp312-cp312-win_amd64.whl (1.2 MB)
    1.2/1.2 MB 2.3 MB/s eta 0:00:00
Installing collected packages: psycopg2
Successfully installed psycopg2-2.9.9

[notice] A new release of pip is available: 23.3.2 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\MANASWI\DJANGO\django in one video>
```

2. Update DATABASES Settings: In your Django project's settings.py file, locate the DATABASES setting and update it to use the PostgreSQL database. Replace the placeholders with your actual database details.
3. Apply Migrations: Run the following Django management command to apply migrations and create the necessary database tables → `python manage.py migrate`
4. Verify Connection: To ensure that the connection is working, run the Django development server → `python manage.py runserver`.

6.Add Members

Adding members in a Django application typically involves creating a User model, handling user registration, and managing user authentication. Django provides a built-in authentication system that you can use for this purpose. Here's a step-by-step guide:

1. **Create a User Model:** In your Django app, create a custom User model by extending the AbstractUser class provided by Django. This allows you to add additional fields to the default User model.
2. **Run Migrations:** After creating the custom User model, run the following commands to apply the migrations→`python manage.py makemigrations python manage.py migrate`
3. **User Registration:** Create views and templates for user registration. You can use Django's built-in UserCreationForm for registration forms.
4. **Login and Logout Views:** Django provides built-in views for login and logout. You can use these views or create custom views if needed.
5. **Secure URLs:** Protect views that require authentication using the `@login_required` decorator.
6. **User Authentication in Templates:**Use `{% if user.is_authenticated %}` in templates to conditionally display content based on whether the user is authenticated.
7. **Password Reset:** Django provides built-in views and templates for password reset functionality. You can customize these as needed.
8. **Secure Settings:** Always secure your settings, especially SECRET_KEY and sensitive information. Use environment variables or a configuration management tool.
9. **Additional Considerations:** Implement user profile pages if needed→Customize the Django admin to manage users.

7. Deploy Django

1. Set Up a Production Environment:

1. Server Hosting: Choose a server hosting provider. Popular choices include AWS, DigitalOcean, Heroku, and others.
2. Domain Name: Purchase a domain name from a domain registrar. This will be used to access your deployed application.
3. Server Configuration: Configure your server with necessary software (Python, PostgreSQL/MySQL, etc.) and system libraries. Set up security measures like firewalls and SSH key authentication.

2. Configure Django for Production:

1. Update settings.py:
 - Set `DEBUG = False`.
 - Add your server's IP address or domain to `ALLOWED_HOSTS`.
 - Configure your database settings for the production environment.
2. Static Files: Set up a proper mechanism for serving static files. You can use WhiteNoise, Django Whitenoise, or configure your web server to serve them.
3. Media Files: If your application involves user-uploaded files, configure a storage backend like Amazon S3.

3. Web Server Configuration:

1. Choose a Web Server: Common choices are Nginx or Apache. Gunicorn or uWSGI can be used as application servers.
2. Install and Configure the Web Server: Install and configure the web server to proxy requests to your Django application. Set up static file serving.

4. Set Up SSL/TLS:

1. Obtain SSL Certificate: Secure your application by obtaining an SSL certificate. You can use services like Let's Encrypt for free certificates.
2. Configure SSL in the Web Server: Update your web server configuration to use SSL/TLS. Redirect HTTP traffic to HTTPS.

5. Deploy the Database:

1. Migrate Database: Run migrations on your production database → `python manage.py migrate`

6. Set Up Process Manager:

1. Choose a Process Manager: Gunicorn or uWSGI can be used to manage your Django application process.
2. Install and Configure the Process Manager: Install and configure the process manager to keep your Django application running.

7. Deploy Your Application:

1. Upload Your Code: Upload your Django code to the server. This could be done through version control (Git) or other means.
2. Install Dependencies: Install the required dependencies by running `pip install -r requirements.txt`
3. Collect Static Files: Collect static files using `python manage.py collectstatic`
4. Restart Application Server: Restart your application server to apply changes.

8. Monitor and Maintain:

1. Logging: Set up logging to monitor errors and issues.
2. Monitoring Tools: Use monitoring tools like New Relic, Sentry, or others to keep an eye on application performance.
3. Automate Deployment: Consider using deployment automation tools like Fabric or Ansible.

9. Testing:

1. Test the Deployment: Thoroughly test your deployed application to ensure everything is working as expected.

10. Continuous Integration/Continuous Deployment (CI/CD):

1. Set Up CI/CD Pipeline: Automate testing and deployment with CI/CD tools like Jenkins, GitLab CI, or GitHub Actions.

11. Scale:

1. Horizontal Scaling: If needed, set up load balancing and consider scaling horizontally by adding more server instances.

9.Elastic Beanstalk (EB)

1. Create requirements.txt:

The requirements.txt file lists all the Python packages and their versions that your Django application depends on. It's a standard practice to include this file in your project to easily manage dependencies. You can generate it by running → `pip freeze > requirements.txt`

2. Create django.config:

The django.config file is used to configure Elastic Beanstalk settings specific to Django applications. It might contain configurations such as setting environment variables, specifying the WSGI server, or other Django-specific configurations.

3. Create .zip File:

Create a zip file containing your Django project files, requirements.txt, and django.config. Exclude any files that are unnecessary for deployment, such as virtual environment directories, local settings, and other development-specific files → `zip -r your_project.zip your_project -x "your_project/venv/*" -x "your_project/.git/*" -x "your_project/local_settings.py"`

4. Deploy with EB:

- Log in to the AWS Management Console and navigate to Elastic Beanstalk.
- Create a new environment.
- Choose the web server environment that matches your Django application (Python).
- Upload the .zip file you created in the previous step.
- Configure environment options, such as database settings, instance type, and more.
- Review and confirm the settings, then click "Create environment."

5. Update Project: When you make changes to your Django application, update the project by: Modifying your code → Updating requirements.txt if new dependencies are added → Creating a new .zip file → Uploading the new .zip file to your Elastic Beanstalk environment using the Elastic Beanstalk console or the EB CLI. EB will automatically deploy the changes and update your running environment.

More Django

Add Slug Field: A slug field is often used in Django models to create a URL-friendly representation of a string. To add a slug field, modify your model and include a SlugField().

Add Bootstrap 5: To add Bootstrap 5 to your Django project → Include Bootstrap CSS and JS files in your HTML templates → You can either download Bootstrap files and include them locally or use a CDN.

Conclusion

Django stands out as a powerful and versatile web framework that has gained widespread popularity in the development community. Django offers a robust set of tools and conventions to streamline the creation of web applications. One of Django's key strengths lies in its built-in features, such as an Object-Relational Mapping (ORM) system, authentication, and an admin interface. These components significantly reduce development time and make it easier for developers to build scalable and maintainable applications. Django follows the "Don't Repeat Yourself" (DRY) and "Convention Over Configuration" (CoC) principles, promoting clean and organized code. The framework's extensive documentation and an active community contribute to its accessibility, providing ample resources for developers to troubleshoot issues and enhance their skills. Django's support for various databases, its adherence to best practices, and the availability of reusable apps in the Django ecosystem make it a suitable choice for projects of different scales and complexities.