Introduction to Django

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

Ridiculously fast:
Django was designed to help developers take applications from concept to completion as quickly as possible.

Reassuringly secure:
Django takes security seriously and helps developers avoid many common security mistakes.

Exceedingly scalable:
Some of the busiest sites on the web leverage Django's ability to quickly and flexibly scale.

Why Django?

With Django, you can take web applications from concept to launch in a matter of hours. Django takes care of much of the

hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

Ridiculously fast.
Django was designed to help developers take applications from concept to completion as quickly as possible.

Fully loaded.
Django includes dozens of extras you can use to handle common web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds, and many more tasks — right out of the box.

Reassuringly secure.
Django takes security seriously and helps developers avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery, and clickjacking. Its user authentication system provides a secure way to manage user accounts and passwords.

Exceedingly scalable.

Some of the busiest sites on the planet use Django's ability to quickly and flexibly scale to meet the heaviest traffic demands.

Incredibly versatile.
Companies, organizations, and governments have used Django to build all sorts of things—from content management systems to social networks to scientific computing platforms.

Install Django

Step 1: Install Python

Most Ubuntu systems come with Python pre-installed, but it's a good practice to install the latest version or the version you prefer.

1. Update your package list:

sudo apt update

2. Install Python 3:

sudo apt install python3

3. Verify the installation:

```
python3 --version
```

Step 2: Install pipx

`pipx` is a tool designed to help you install and run Python applications in isolated environments. It simplifies the management of Python packages, especially those that are command-line applications, by creating a separate virtual environment for each application.

1. Install `pipx`:

```
sudo apt install pipx
```

2. Confirm the pipx installation

```
pipx --version
```

Step 3: Install Django

1. Install Django using `pipx`:

```
pipx install django
```

2. Show pipx list Confirm the installation

```
pipx list
```

Step 4. Always Activate the

Virtual Environment on Startup

1. Open the configuration file with a text editor (e.g., for Bash):

nano ~/.bashrc

2. Add the line last on this file:

source /home/coderaktar/.local/share/pipx/venvs/django/bin/activate

coderaktar => Your Ubuntu/OS username

3. Save the changes:

In nano, press , then to save, and to exit.

4. Reload the configuration:

To apply the changes immediately, run:

source ~/.bashrc

Now Your Machine is ready for creating Django project.

Create django project

1. Go to the Directory where you create project.

2. Then open terminal on this Directory

3. Write in the terminal this command:

```
django-admin startproject My_First_Project
# Here My_First_Project is your project name
```

5. After execute this command we can see a Directory named My_First_Project[Your give project name]

6. Open the Directory where we see a file named manage.py and a Directory same as our project name

7. Open terminal again on this folder or write the previous terminal goes for My_First_Project Directory:

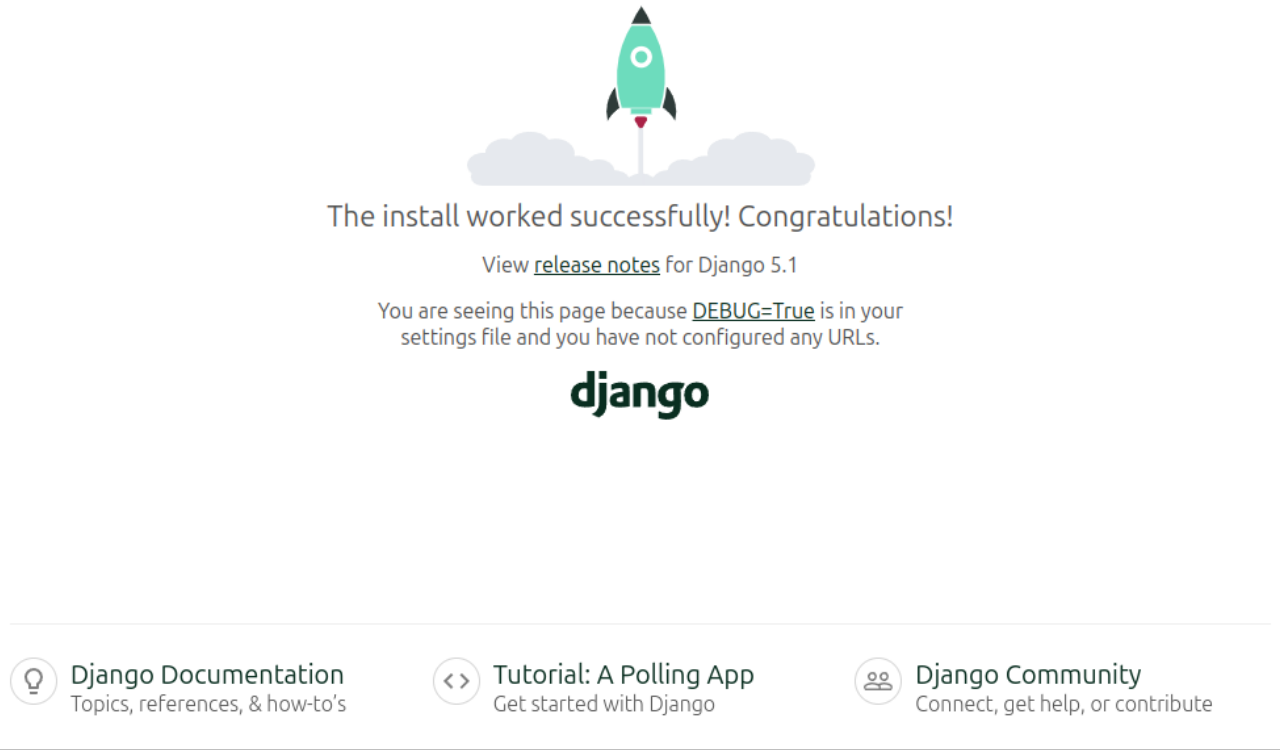
```
cd My_First_Project/
```

8. Navigate to your Django project directory and run:

```
python manage.py runserver
```

9. Now you see a server address ctrl + click **to open it or copy it and open a browser past and hit the link**

10. If all set up successfully completed you can see the page on this link: Congratulation



Run Our Application is Specific Port

All procedure are same. When we run application in the server open it specific port:

Syntax:
python manage.py runserver protnumber

Example:
python manage.py runserver 7000

Django Application

Django Application Overview:

• A **Django project** is a collection of applications and configurations.

• Combining multiple applications creates the full Django website.

• Each application within a project typically serves a specific purpose, such as **user registration**, **blog posting**, **commenting**, etc.

Command to Create a New App:

To create a new app in your Django project, use the following command:

```
python manage.py startapp first_app
```

Connecting the App to the Project:

1. Open the `settings.py` file located in your project's main folder.

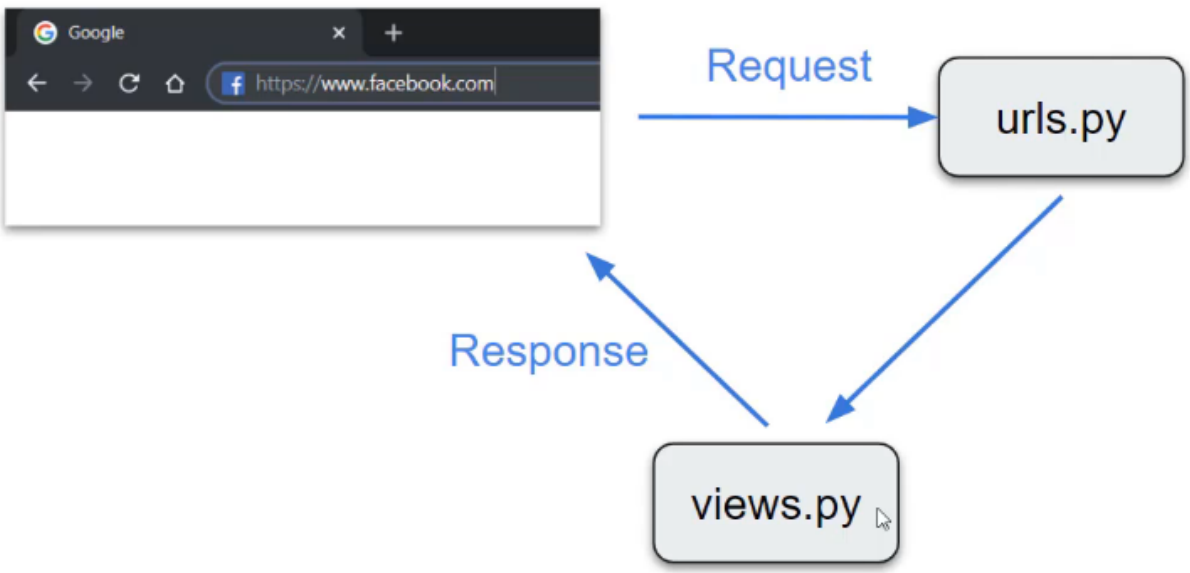
2. Inside `settings.py`, you'll find a list called `INSTALLED_APPS`.

3. To connect the newly created app to the project, add the app's name (`first_app`) to this list as an item:

```
INSTALLED_APPS = [
    # Other installed apps...
    'first_app',
]
```

View, URLs, Templete

View and URL



Understanding URL and View in Django

• **URL**: The URL is what we type in the browser's address bar (e.g., `http://127.0.0.1:8000/index/`).

• **View**: The view refers to what is displayed on the webpage after you visit a particular URL. In Django, a view is a function that processes the request and returns a response (usually an HTML page or other content).

View and URL

How URL and View Work Together in Django:

1. When you type a URL in the browser, it sends a **request** to the Django application.
2. Django checks the `urls.py` file to match the requested URL with a defined path.
3. If a match is found, the corresponding **view** function is called from the `views.py` file.
4. The view function processes the request and returns a response (e.g., an HTML page) that is displayed in the browser.

Steps to Implement a URL and View in Django:

1. **Create an Application**: First, ensure that you have an app created in your Django project. In this case, letâ€™s assume you created an app named `first_app`.

2. **Create a View**: A view is a Python function that takes an HTTP request as an argument and returns a response. You need to define this function in `views.py` inside the `first_app` folder.

- Import `HttpResponse` to handle the response from URLs:

Create a view called `index` in `views.py`:

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

- Create a view called `index` in `views.py`:

```
def index(request):
    return HttpResponse("Bismillahir Rahmanir Rahim")
```

Here, the `index` function takes a `request` parameter and returns the message **“Bismillahir Rahmanir Rahim”** as the response. This message will be displayed when the view is accessed through the browser.

3. **Connect the View to a URL**: To display the view when a URL is accessed, you need to configure it in `urls.py`. Here's how:

◇ Open the `urls.py` file in your project folder.

◇ Import the views from your `first_app`:

```
from first_app import views
```

◇ Define a path in the `urlpatterns` list:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index', views.index, name='index'),
]
```

Explanation:

• **First parameter (path)**: `'index/'` is the URL pattern, meaning when you visit `http://127.0.0.1:8000/index/`, this URL will match.

• **Second parameter (view name)**: `views.index` refers to

the view function (`index`) defined in `views.py`.

• **Third parameter (name)**: `name='index'` is an optional name for the URL pattern, which can be useful for reverse URL lookups in templates or vie

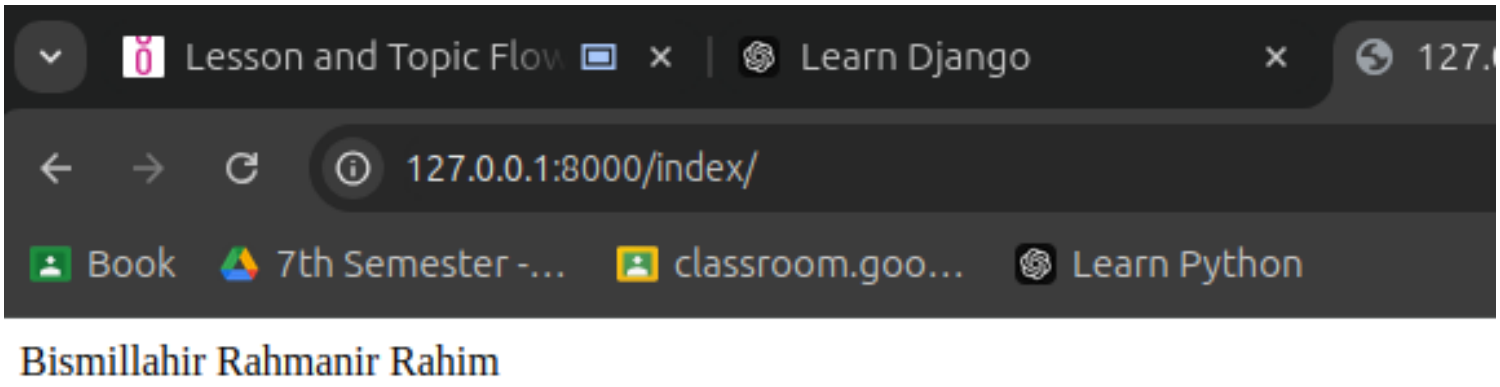
4. **Viewing the Result in the Browser**:

◇ After configuring the URL, run the server:

```
python manage.py runserver
```

◇ In the browser, go to: `http://127.0.0.1:8000/index/`

◇ The message **“Bismillahir Rahmanir Rahim”** will appear.



Adding HTML Tags in Views:

You can also include HTML in the response returned by a view.

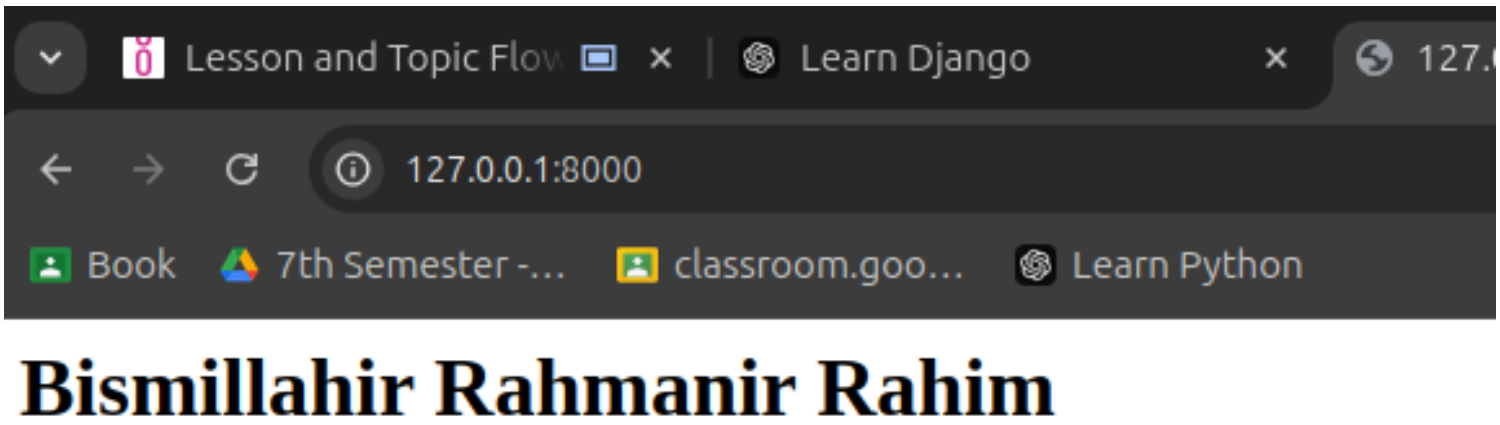
For example, to make the text appear as a header:

```
def index(request):
    return HttpResponse("<h1>Bismillahir Rahmanir Rahim</h1>")
```

Default URL (Empty Path):
If you want this view to be shown when the base URL (i.e., `http://127.0.0.1:8000/`) is visited, you can update the path in `urls.py` to an empty string:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='index'),
]
```

Now, visiting `http://127.0.0.1:8000/` will display the `index` view.



Creating and Linking Multiple Pages in Django Using Views and URL Patterns

In Django, each view corresponds to a webpage, and URLs are the paths that users type in their browser to access different views.
Let's break down the examples:

1. Creating Views in `views.py`

In the `views.py` file, you create functions (views) that return an `HttpResponse`. These responses contain HTML that is rendered in the browser.

Home Page View (index):

- This view displays the **Home Page** and contains links to the **About** and **Contact** pages using `<a>` tags (hyperlinks).
- The links direct users to `127.0.0.1:8000/about/` and `127.0.0.1:8000/contact/` respectively.

```
def index(request):
    return HttpResponse("<h1>Home Page</h1> <a href='about/'>About Page</a> <a href='contact/'>Contact Page</a>")
```

- `href='about/'`: This link will take the user to the **About**

Page (127.0.0.1:8000/about/).

- `href='contact/'`: This link will take the user to the **Contact Page** (127.0.0.1:8000/contact/).

About Page View (about):

- This view displays the **About Page** and includes links back to the **Home Page** and the **Contact Page**.

```
def about(request):
return HttpResponse("<h1>About Page</h1> <a href='/'>Home Page</a> <a href='/contact/'>Contact Page</a>")
```

- `href='/'`: This link directs the user back to the **Home Page** (127.0.0.1:8000/).
- `href='/contact/'`: This link directs the user to the **Contact Page** (127.0.0.1:8000/contact/).

Contact Page View (contact):

- This view displays the **Contact Page** and includes links to the **Home Page** and the **About Page**.

```
def contact(request):
    return HttpResponse("<h1>Contact Page</h1> <a href='/'>Home Page</a> <a href='/about/'>About Page</a>")
```

• `href='/'`: This link directs the user back to the **Home Page** (`127.0.0.1:8000/`).

• `href='/about/'`: This link directs the user to the **About Page** (`127.0.0.1:8000/about/`).

2. Mapping URLs in `urls.py`

In the `urls.py` file, we map URLs to the views we created. This is where Django connects the URL path that users type in the browser to the appropriate view.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='index'),
    path('about/', views.about, name='about'),
    path('contact/', views.contact, name='contact'),
]
```

• `path('', views.index, name='index')`:

◇ This path is for the **Home Page**.

◇ The empty string (`''`) means that this is the default page (`127.0.0.1:8000/`).

◇ The `views.index` specifies that when this URL is visited, the `index` view will be called.

◇ `name='index'` gives this URL pattern a name for easy reference in templates or other views.

• `path('about/', views.about, name='about')`:
    ◇ This URL pattern maps to the **About Page** (`127.0.0.1:8000/about/`).
    ◇ When this URL is accessed, the `about` view will be called to display the content.

• `path('contact/', views.contact, name='contact')`:
    ◇ This URL pattern maps to the **Contact Page** (`127.0.0.1:8000/contact/`).
    ◇ When this URL is accessed, the `contact` view will be called to display the content.

URL Mappings

URL Mappings in Django: Structuring URLs for Multiple Applications

When building a Django project with multiple apps, managing a large number of views can become cumbersome if all the paths are defined in the main `urls.py` file of the project. Instead, Django allows us to organize URLs by

creating separate `urls.py` files in each app, and then including them in the project's `urls.py`. This practice is called **URL mapping** and helps to keep the project organized and scalable.

Problem:

• Suppose you have 10 applications, and each app has 10 views. This would result in 100 views (10 apps × 10 views = 100 views). Defining all the paths for these views in the main `urls.py` file would not be a good practice because it becomes difficult to manage.

Solution: URL Mapping with `include()`

You can create a `urls.py` file inside each app to define the app-specific URL patterns. Then, in the main `urls.py` file of the project, use the `include()` function to link the app's URLs to the project's URLs.

Step-by-Step Process

1. Create Views in `views.py` of Your Application

In your Django application (e.g., `first_app`), you define the views that handle requests and return responses. Let's create two views in `first_app/views.py`:

```
from django.shortcuts import render
```

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("<h1>Home Page</h1>")
def contact(request):
    return HttpResponse("<h1>Contact Page</h1>")
```

Here, we've created two simple views:
• `index`: Displays a "Home Page".
• `contact`: Displays a "Contact Page".

2. Create a `urls.py` File in the Application (first_app)

Next, create a new `urls.py` file inside the `first_app` directory. This file will define the URL patterns specific to the `first_app`.

```
# first_app/urls.py
from django.urls import path
from first_app import views


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

• `path('', views.index, name='index')`: This sets the base URL (empty string `''`) for the **Home Page**. Visiting `127.0.0.1:8000/customer/` will display the **Home Page**.

• `path('contact/', views.contact, name='contact')`: This sets the URL for the **Contact Page**. Visiting `127.0.0.1:8000/customer/contact/` will display the **Contact Page**.

3. Modify the Project's Main `urls.py` File

Now, in the main `urls.py` file of the project (usually located in the root folder of the project), you will include the URL patterns from the `first_app`.

```
# First_Project/urls.py
from django.contrib import admin
from django.urls import path, include


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

`include('first_app.urls')`: This tells Django to include all the URL patterns defined in `first_app/urls.py` when the base URL is prefixed with `customer/`.• For example, visiting `127.0.0.1:8000/customer/` will load the Home Page (`index`

view), and `127.0.0.1:8000/customer/contact/` will load the Contact Page (`contact` view).

How URL Mapping Helps:

• By using `include()`, you are separating URL patterns for each app, making your code cleaner and more modular.

• Instead of defining all URLs in the main `urls.py`, you define URLs specific to an app inside the appâ€™s own `urls.py`.

• This approach scales well with large projects, allowing you to add and manage views for each app without cluttering the main URL configuration.

Template

Django Templates

A **template** is a file where you describe how the result of your web page should be presented. It is typically an HTML file, but Django allows logic to be added using **Django Template Language (DTL)**, which includes special tags and placeholders.

For example, you can use placeholders like `{{ firstname }}` to dynamically display data on the page:

```
    <h1>My Homepage</h1>
    <p>My name is {{ firstname }}.</p>
```

Steps to Work with Templates in Django

1. Set Up the Template Directory

To use templates, you first need to create a folder where your HTML files (templates) will be stored. This folder is generally named `templates` and is placed at the project level (where `manage.py` is located).

In our example, the project has an app called `first_app`, so we'll organize the templates as follows:

- Create a folder named `templates` in the project root (where `manage.py` is).
- Inside the `templates` folder, create another folder named `first_app` (for organizing templates specific to the `first_app`).
- Inside the `first_app` folder, create an HTML file named `index.html`.

```
First_Project/
│
├── First_Project/
├── first_app/
├── templates/
│   └── first_app/
```

```
│       └── index.html
└── manage.py
```

2. Create the `index.html` Template

The `index.html` file will contain the HTML code that defines the layout of the page.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home Page</title>
</head>
<body>
    <h1>This is the first homepage</h1>
</body>
</html>
```

3. Configure the Template Directory in `settings.py`

Next, you need to inform Django where to find the templates.

1. In your project's `settings.py` file, add the path to the `templates` directory using `BASE_DIR`:

```
BASE_DIR = Path(__file__).resolve().parent.parent
```

```
TEMPLATES_DIR = Path(BASE_DIR).joinpath('templates')
```

2. Add the `TEMPLATES_DIR` to the `TEMPLATES` list in `settings.py`. The `DIRS` key should hold a list of directories where Django will search for templates:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATES_DIR,],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

3. For confirmation, you can print the `TEMPLATES_DIR`:
print(TEMPLATES_DIR)

4. Create a View in `views.py`

Now, you need to create a view in `first_app/views.py` that will render the `index.html` template and pass data to it if needed.

1. In `views.py`, import the `render()` function.
2. Define a dictionary (e.g., `dict`) that contains the data to be passed to the template.
3. Use `render()` to return the HTML page, specifying the template location and the data.

```
from django.shortcuts import render

def index(request):
    context_dict = {'name': 'Bismillahir Rahmanir Rahim'}  # Data to be passed to the template
    return render(request, 'first_app/index.html', context=context_dict)  # Rendering the template
```

5. Display Dynamic Content in `index.html`

To use the data passed from the view, such as the `name` key, you can include it in the `index.html` file by using Djangoâ€™s template tags.

In `index.html`, update the content like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
```

```
initial-scale=1.0">
    <title>Home Page</title>
</head>
<body>
    <h1>Welcome to the First Homepage</h1>
    <p>{{ name }}</p>  <!-- Display the dynamic content passed
from the view -->
</body>
</html>
```

• `{{ name }}`: This is a Django template tag that inserts the value of the `name` variable (which was passed in the `context_dict`) into the HTML.

6. Update the `urls.py` to Route to the View

Make sure that the `index` view is connected to a URL pattern in `first_app/urls.py`. If you donâ€™t have `urls.py` in your app, create it:

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

urlpatterns = [
    path('', views.index, name='index'),  # The index view for the
home page
]
```

And include it in the projects main `urls.py`:

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

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

Now, visiting `http://127.0.0.1:8000` will render the `index.html` page and display the dynamic content passed from the view.

Static Files

Handling Static Files in Django

When building web applications, you will often need to include static files such as CSS, JavaScript, and images. In Django, static files are handled separately from dynamic content to improve organization and performance. Follow

these steps to set up and use static files effectively in a Django project.

Project Structure Example

Assume you have a Django project called `First_Project` with an app called `first_app`. Here's how you should organize your project to manage static files.

```
First_Project/
│
├── First_Project/          # Main project folder
├── first_app/              # App folder
├── static/                 # Static folder at the root of the project
│   └── first_app/          # App-specific static folder
│       ├── images/         # Folder for images
│       │   └── image.jpg   # Example image file
│       └── css/            # Folder for CSS
│           └── style.css   # Example CSS file
├── templates/              # Templates folder
│   └── first_app/          # App-specific templates folder
└── manage.py               # Main project management script
```

Step-by-Step Setup

1. Create the `static` Directory

1. **At the project root** (where `manage.py` is located), create a folder named `static`.
2. Inside the `static` folder, create another folder named after

your app (in this case, `first_app`). This is useful for organizing static files specific to each app.
3. In the `first_app` folder:• Create a folder called `images` to store your image files.
• Create another folder called `css` to store your CSS files.

This setup ensures that all static files are organized by app and file type.

2. Configure Django to Find Static Files

In your `settings.py` file, you need to define paths so Django knows where to find static files.
◇ First, import `Path` from `pathlib` to easily work with file paths.
◇ Define paths for the base directory (`BASE_DIR`), templates, and static files.

```
from pathlib import Path

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
TEMPLATES_DIR = Path(BASE_DIR).joinpath('templates')
STATIC_DIR = Path(BASE_DIR).joinpath('static')
```

Next, configure the static files settings in `settings.py`:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
```

```
    STATIC_DIR,
]
```

3. Load Static Files in HTML Templates

To load static files (like CSS and images) in your HTML templates, follow these steps:

1. Add `{% load static %}` at the top of your HTML files. This Django template tag allows you to reference static files.
2. Use the `{% static %}` tag to load images and CSS files correctly.

Here's how it looks in a sample HTML file:

Loading an Image

In `templates/first_app/index.html`, add the following code to load an image from the `static/first_app/images/` directory:

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home Page</title>
</head>
<body>
    <h1>Welcome to the First Homepage</h1>
```

```
    <p>{{ name }}</p>  <!-- Display the dynamic content passed from the view -->
    <br>
    <img src="{% static "first_app/images/image.jpg"%}" alt="Something Problem">
</body>
</html>
```

Linking a CSS File

You can link a CSS file from the `static/first_app/css/` directory like this:

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home Page</title>
</head>
<body>
    <h1>Welcome to the First Homepage</h1>
    <p>{{ name }}</p>  <!-- Display the dynamic content passed from the view -->
    <br>
    <img src="{% static "first_app/images/image.jpg"%}" alt="S-
```

```
omething Problem">
</body>
</html>
```

In the above code:
• `{% load static %}` enables the use of static files.
• `{% static 'path_to_file' %}` is used to reference the static file path.

Relative URLs

Using Relative URLs in Django with `url` Template Tag

In Django, you can use the `{% url %}` template tag to create links by referring to the **name** of the URL pattern defined in `urls.py` rather than writing hardcoded URLs. This practice reduces errors and makes URLs easier to manage when they change, as Django automatically resolves them.

Setting up `urls.py` for the App

1. **Define the `app_name` in your `urls.py`:**This allows you to refer to the URL patterns by name within the context of the app.

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

app_name = 'first_app'

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

• `app_name = 'first_app'` allows you to use URL names like `'first_app:index'` and `'first_app:about'` in your templates.
• Each `path()` in `urlpatterns` has a **name** parameter, which makes referencing these views easier in templates.

HTML Templates Using `{% url %}`

Once you've defined the named URL patterns, you can use the `{% url %}` tag in your templates to link between different pages. Here are examples of how to do this.

1. `index.html` (Home Page)

This template uses the `{% url %}` tag to create a link to the

"About" page by referring to the named URL `'first_app:about'`.

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="{% static 'first_app/css/style.css' %}">

    <title>Home Page</title>
</head>
<body>
    <h1>Welcome to the First Homepage</h1>
    <p>{{ name }}</p>  <!-- Display the dynamic content passed from the view -->
    <br>
    <img src="{% static "first_app/images/image.jpg"%}" alt="Something Problem">
    <br>
    <a href="{% url 'first_app:about' %}">About Page</a>

</body>
</html>
```

2. `about.html` (About Page)

This template links back to the "Home" page using the named URL `'first_app:index'`.

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="{% static 'first_app/css/style.css' %}">
    <title>About Page</title>
</head>
<body>
    <h1>This is about page.</h1>
    <a href="{% url 'first_app:index'  %}">Home Page</a>
</body>
</html>
```

Explanation:

1. **Loading Static Files:**

• `{% load static %}` is used to enable Django's static file handling in your templates.

• Static files (CSS, images, JavaScript) are referenced using `{% static 'path/to/file' %}`.

2. **Using the `{% url %}` Tag:**
• `{% url 'first_app:about' %}` generates a URL that points to the view named `about` within the app named `first_app`.
• Similarly, `{% url 'first_app:index' %}` generates a URL to the home page (`index` view).

This way, the links within your templates are dynamic and maintainable. If you ever change the URL structure in `urls.py`, the templates won't break as long as you update the URL names.

Models

Django Models Overview

In web development, databases are essential for storing and retrieving data. Django, as a web framework, provides built-in support for working with databases. By default, Django uses **SQLite** as its built-in database for development, which is sufficient for small or simple projects. For larger, more complex projects, databases like **MySQL** or **MongoDB** are often used.

Working with Databases in Django: Models

To interact with a database in Django, you use **models**. A model defines the structure of your database by specifying the data fields and behaviors. Each model typically corresponds to a single table in the database.

Key Points About Django Models:

• **A model is a Python class** that subclasses `django.db.models.Model`.

• Each **attribute** of the model class represents a **field** in the database.

• **Django automatically generates the database access API** for the model, making it easy to query and manipulate data.

Steps to Use Models:

1. **Locate `models.py`:**

◇ Each Django app has a `models.py` file where models are defined.

◇ Open this file in the app where you want to define the models.

2 **Import Models Library:**

◇ The models library is already imported by default in `models.py`:

```
from django.db import models
```

Defining a Model:

A model in Django is defined as a Python class. Here's a basic example that defines a `Person` model with two fields: `first_name` and `last_name`.

```
from django.db import models

# Create your models here.
class Person(models.Model):
    first_name = models.CharField(max_length=30)  # A
character field with a max length of 30
    last_name = models.CharField(max_length=30)
```

In this example:

• The `Person` class inherits from `models.Model`, which is required for all models.

• `first_name` and `last_name` are defined as **fields** using Djangoâ€™s built-in `CharField` to store character data with a maximum length of 30.

Database Table Creation:

When the `Person` model is created, Django will generate the SQL to create a table for this model. For example, the following SQL would be generated to create the `Person` table:

```
CREATE TABLE myapp_person (
    "id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAUL-
T AS IDENTITY,
```

```
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

• Django automatically adds an `id` **field** as the primary key unless you specify one yourself.
• The `first_name` and `last_name` fields map directly to columns in the table.

Summary:
• **Models** in Django define the structure of your database.
• Each model maps to a database table, and each model field maps to a column in that table.
• Django provides an **automatic API** for interacting with the database, making data management easier.

This approach helps keep your data organized and easily accessible within your Django application.

Creating Models

Creating Models in Django

First, you need to **import the models module** from Django to define your database structure:

```
from django.db import models
```

Creating the `Musician` Model:

We define a model (which corresponds to a database table) called `Musician`. Each model field is a column in the table.

```
class Musician(models.Model):
    name = models.CharField(max_length=50)          # Field for the musician's first name (character limit of 50)
    last_name = models.CharField(max_length=50)     # Field for the musician's last name (character limit of 50)
    instruments = models.CharField(max_length=100)  # Field for the musician's instrument(s) (character limit of 100)
```

Auto Primary Key: By default, Django adds an **auto-incrementing primary key** field called `id` to each model, so you don't need to explicitly declare it unless you want to use a custom field for the primary key.

Creating the `Album` Model:

Next, we define an `Album` model, which references the `Musician` model using a **ForeignKey** field:

```
class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CA-
```

```
SCADE)  # ForeignKey linking to Musician model
    name = models.CharField(max_length=100)                    #
Album name (character limit of 100)
    release_date = models.DateField()                          # Album
release date
    num_stars = models.IntegerField()                          #
Number of stars (integer rating)
```

• `artist = models.ForeignKey(Musician, on_delete=models.CASCADE)`: The `artist` field creates a relationship between the `Album` and `Musician` models. The `on_delete=models.CASCADE` means that if the associated `Musician` record is deleted, the related `Album` records will also be deleted.

Explanation of Fields:

1. `models.CharField(max_length=50)`: A string field for storing text (e.g., name) with a maximum length.
2. `models.IntegerField()`: A field for storing integers (e.g., number of stars).
3. `models.DateField()`: A field for storing dates (e.g., release date).
4. **ForeignKey**: A field that establishes a many-to-one relationship between two models (e.g., linking an album to a musician).

Auto Primary Key in Django Models:

◇ Django automatically creates an `id` field as the primary key for each model unless you explicitly define a different primary key.

◇ Even though you havenot declared `id` in the `Musician` model, Django adds it by default, which is why it can be used as a **ForeignKey** in the `Album` model.

Converting Models to SQL Tables

Once you've defined your models, you need to convert them into SQL database tables. Here's how to do it:

1. **Run initial migrations**:

```
python manage.py migrate
```

This command applies Django's built-in migrations, such as creating the initial database schema (including tables for authentication, sessions, etc.).

2. **Create migrations for the new `Musician` and `Album` models**:

```
python manage.py makemigrations musician
```

This command tells Django that you have made changes (defined new models) and creates migration files for the `Musician` and `Album` models.

3. **Apply the new migrations**:

python manage.py migrate

This command applies the migrations to the database, creating the corresponding tables (`Musician` and `Album`).

Explanation of Commands:

1. `python manage.py migrate`:

→ This command applies all available migrations to the database, setting up the tables for any installed apps and Django itself.

• `python manage.py makemigrations musician`:

→ This command detects changes in your models (like creating `Musician` and `Album`) and prepares a migration file, which is a Python file that contains instructions to modify the database.

• `python manage.py migrate`:

→ After creating the migration files, this command applies them, making the changes in the database by creating or altering tables accordingly.

These steps ensure that your models are reflected in the database as actual SQL tables with the appropriate fields and relationships.

Model Objects

Adding Data to the Musician Table in Django

In Django, models represent database tables, and each **instance (object)** of a model represents a **row** in the corresponding database table. To add entries to the **Musician** table, you follow these steps:

1. Open the Django Shell

The Django shell allows us to interact with the Django environment. We use the shell provided by `manage.py` because it loads Django's settings, including the database configuration

```
python manage.py shell
```

2. Import the `Musician` Model

After opening the shell, import the `Musician` model from your app:

```
from musician.models import Musician
```

3. Check Existing Entries

To view the current entries in the **Musician** table, use the

following command:
print(Musician.objects.all())

Initially, it returns an empty query set because there are no entries yet:
<QuerySet []>

4. Add a New Entry

To add an entry to the **Musician** table, create an instance of the `Musician` class with the required fields (name, last_name, instruments), and then save it to the database.

```
obj = Musician(name="Eric", last_name="Clapton", instruments="Guitar")
obj.save()  # Save the object to the database
```

You can add more entries by creating new objects and saving them in the same way:

```
obj = Musician(name="Hilary", last_name="Han", instruments="Violin")
obj.save()
```

5. View the Entries Again

Now, when you run the command to retrieve all entries again:
print(Musician.objects.all())

It will return a **QuerySet** with the entries youâ€™ve added,

but they are displayed as generic `Musician` objects:
<QuerySet [<Musician: Musician object (1)>, <Musician: Musician object (2)>]>

6. Improving the Output Display

To make the output more readable, you can add a method in the `Musician` model to display the entries in a user-friendly way. In `models.py`, add the `__str__` method:

```
class Musician(models.Model):
    name = models.CharField(max_length=50)          # Field for the musician's first name (character limit of 50)
    last_name = models.CharField(max_length=50)     # Field for the musician's last name (character limit of 50)
    instruments = models.CharField(max_length=100)  # Field for the musician's instrument(s) (character limit of 100)

    def __str__(self):
        return self.name + " " + self.last_name + " " + self.instruments
```

The `__str__` method ensures that when you print objects, they will display more informative text.

7. Apply Migrations for the Changes

Since you modified the model, you need to run migrations again to apply the changes:

```
python manage.py makemigrations musician
python manage.py migrate
```

8. Reopen the Shell and Check the Entries

After migrating, reopen the Django shell, import the model again, and check the entries:

```
python manage.py shell

from musician.models import Musician
print(Musician.objects.all())
```

Now, the output will be more readable, showing the musician's full name and instrument:

```
<QuerySet [<Musician: Eric Clapton Guitar>, <Musician: Hilary Han Violin>]>
```

Summary of Commands:

1. **Open Django shell**:

```
python manage.py shell
```

2. Import the model:

```
from musician.models import Musician
```

3. Create an entry and save:

```
obj = Musician(name="Eric", last_name="Clapton",
```

```
instruments="Guitar")
obj.save()
```

4. Check entries:

```
print(Musician.objects.all())
```

5. Modify the model (`__str__`) for better display:

```
def __str__(self):
    return self.name + " " + self.last_name + " " +
self.instruments
```

6. Run migrations:

```
python manage.py makemigrations musician
python manage.py migrate
```

By following these steps, you can easily add entries to your Django models and view them in a more readable format.

Admin Site

Working with the Django Admin Panel

Django provides a built-in **admin panel** to make it easier to manage your database. From the admin panel, you can insert, update, delete, and view your data with a user-friendly interface.

1. Register Models in the Admin Panel

To use the admin panel for your models, you need to register them in the **admin.py** file of your app.

- Open **admin.py** of your app (in this case, `musician`).
- Import the models you want to manage in the admin panel:

```
from django.contrib import admin
from musician.models import Musician, Album
```

- Register the models with the Django admin site:

```
admin.site.register(Musician)
admin.site.register(Album)
```

This makes the `Musician` and `Album` models accessible from the admin panel.

2. Create a Superuser

To access the admin panel, you need to create a superuser (an admin user with all permissions).
In the terminal, run the following command:

```
python manage.py createsuperuser
```

You will be prompted to enter some information:

```
Username (leave blank to use 'your_username'):
Email address: example@gmail.com
```

Password:
Password (again):

Once the superuser is created, you can log into the admin panel.

3. Accessing the Admin Panel

Django automatically sets up the admin panel at the **/admin/** URL.

- Start the development server:

  python manage.py runserver

- Go to your browser and visit:

  http://127.0.0.1:8000/admin

- Log in with the superuser credentials you just created.

4. Managing Your Models

After logging in, you will see the **Musician** and **Album** tables listed. From here, you can:

- **Add new records**: Click "Add" next to the model name and fill out the form to create new entries.
- **Edit existing records**: Click on any entry to update its details.
- **Delete records**: Select records and delete them.

5. Admin URL Configuration

The admin panel is accessible because it's included in your project's **urls.py** file. By default, this is already set up when you create a Django project:

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

urlpatterns = [
    # .... other path ....
    path('admin/', admin.site.urls),
]
```

Now, you can manage your models easily through the admin interface.

Summary:

1. **Register models** in **admin.py**:

```
from django.contrib import admin
from musician.models import Musician, Album
admin.site.register(Musician)
admin.site.register(A
```

2. **Create a superuser**:

```
python manage.py createsuperuser
```

3. **Access the admin panel** at:

```
http://127.0.0.1:8000/admin
```

4. **Login** with your superuser credentials to manage your

models.