

DJANGO

VISUAL GUIDE

The Complete Beginner's Handbook

to Mastering Web Development
Effortlessly Without Prior
Programming Experience

2024

DJANGO

VISUAL

GUIDE

The Complete
Beginner's Handbook

D-Libro

Copyright © 2024

Title: Django Visual Guide: Complete beginner's guide

Author: D-Libro Project (a lead author: Ben Bloomfield)

Edition: First Edition (2024)

Publication Date: July 2024

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, please contact the publisher, addressed "Attention: Permissions Coordinator," at the address below.

Alattice OU

Harju maakond, Tallinn, Kesklinna linnaosa, Ahtri tn
12, 15551

info@a-lattice.com

Disclaimer: The information contained in this book is provided for educational and informational purposes only and is not intended to replace the advice of a professional or qualified expert. The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained within this book is presented without warranty, either express or implied.

The author and publisher shall not be held liable or responsible for any loss, injury, or damage caused, directly or indirectly, by the information contained in this book. It is the responsibility of the reader to consult with a professional or qualified expert before making any decisions or taking any actions based on the information provided in this book. The author and publisher do not endorse and expressly deny liability for any product, manufacturer, distributor, service, or service provider mentioned or any opinion expressed in this work.

By reading this book, you acknowledge that you are responsible for your own decisions and actions, and you release the author and publisher from any liability arising from the use of the information contained within this book.

This book is edited based on the Django Introduction course on the D-Libro platform.

<https://d-libro.com/course/django-introduction/>

Table of Contents

[Chapter 01 Django Key Concepts](#)

[Web Framework and Django](#)

[Websites vs. Django Web Apps](#)

[How Django Handles HTTP Request and HTTP Response](#)

[Django's MVT Framework](#)

[Django Templates vs. Django APIs](#)

[Chapter 02 Django Quick Start Guide](#)

[Install Python](#)

[Install Visual Studio Code](#)

[Create Project Directory](#)

[Set Up Virtual Environment](#)

[Install Django](#)

[Start Django Project](#)

[Run Server](#)

[Database Migration](#)

[URL dispatcher – urls.py](#)

[Create Superuser and Log In to Django Admin](#)

[Start App](#)

[Create HTML Templates](#)

[Create Views](#)

[Add URL Patterns](#)

[Project vs. App](#)

[Chapter 03 Django Models and Databases](#)

[Create a Database in Django](#)

[Relational Database](#)

[Create Django Models](#)

[Makemigrations and Migrate](#)

[Add Models in Django Admin – admin.py](#)

[Change Display Name of Record Objects](#)

[Django Models – Data Field Type](#)

[Django Models – Field Options](#)

[Django Models – Help Text Option](#)

[Django Models – Choices Option](#)

[Django Models – DateField with datetime Module](#)

[Django Models – Relationship Fields](#)

[Django Models – ID](#)

[Django Models – ForeignKey\(OneToMany Relationship\)](#)

[Django Models – OneToOneField](#)

[Django Models – ManyToManyField](#)

[Chapter 4 Create CRUD Web Application](#)

[CRUD Web Application](#)

[Basic CRUD Structure in Django](#)

[Django Generic Views](#)

[How To Write Class-Based Views with Generic Views](#)

[Generic View Basic Attributes](#)

[URL Dispatcher for CRUD Views](#)

[Django Templates for CRUD Views](#)

[Django Template Language \(DTL\)](#)

[Template for List Page](#)

[Get_FOO_display method](#)

[Template for Detail Page](#)

[Template with Model Relations](#)

[Template for Create and Update Page](#)

[Template for Delete Page](#)

[Add Links – {% url %} tag](#)

[Extend Templates – {% extends %} tag](#)

[Check Developing App UI on Mobile Device](#)

[Django Templates with Bootstrap](#)

[Crispy Forms](#)

[Customize Views \(1\) – Change List Order](#)

[Customizing Views \(2\) – Filter Lists](#)

[Context](#)

[Customize Views \(3\) – Add Extra Context](#)

[Modularize Templates – {% include %} tag](#)

[Static Files in Development Environment – {% static %} tag](#)

[STATIC_URL and STATICFILES_DIRS](#)

[Create Index HTML](#)

Chapter 5 User Management

User Authentication

Overview of User Management Functions

User Management Function Development with Django

Approaches to Building User Management Functions in Django

Django Allauth (1) – Introduction

Django Allauth (2) – Installation and Initial Settings

Django Allauth (3) – Email Verification via Console

Django Allauth (4) – Email Verification via Gmail

Django Allauth (5) – Social Login with GitHub

Django Allauth (6) – Social Login with Google

Django Allauth (7) – Allauth Template File Setup

Django Allauth (8) – Add Basic Styling with Bootstrap and Crispy Forms

Django Allauth (9) – Customize Sign-in and Sign-up Pages

User Models

Login Required – LoginRequiredMixin

User Login Status Icon on Navigation Bar

Chapter 6 Deploy Django App

Overview of Django App Deployment (1)

Overview of Django App Deployment (2)

Key Steps of Django App Deployment

Hosting Service Initial Settings (1) – AWS Lightsail setup

Hosting Service Initial Settings (2) – SSH Remote Connection

[Manage Local Computer and Remote Server Simultaneously](#)

[Tips for Managing Local Development and Remote Production Environment](#)

[Hosting Service Initial Settings \(3\) – Clone Project Directory with GitHub](#)

[Production Database Setup](#)

[Django Production Settings \(1\) – Settings.py for Development and Production](#)

[Django Production Settings \(2\) – Production Settings](#)

[Django Production Settings \(3\) – django-environ and .env file](#)

[Static File Settings](#)

[Django and Dependency Installation on Production Server](#)

[Web Server and Application Server in Django](#)

[Application Server Setup – Gunicorn](#)

[Web Server Setup – Nginx](#)

[Domain Setup](#)

[SSL Setup – Certbot](#)

[Email Setting – SendGrid](#)

[Social Login for Production](#)

[Manage Local Development and Remote Production Environment](#)

[About D-Libro](#)

Chapter 01

Django Key Concepts

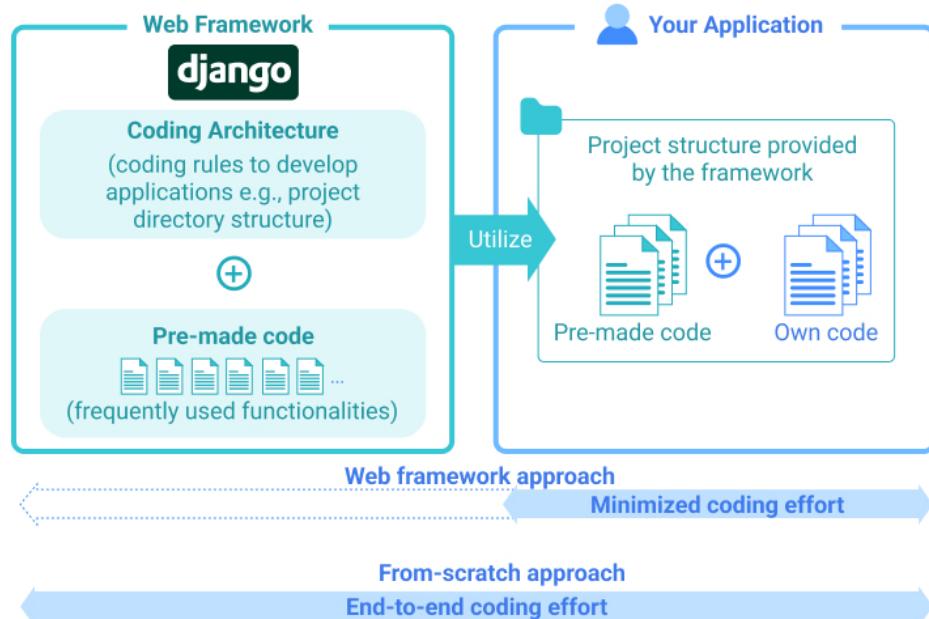
This chapter covers key concepts used in **Django**, a **Python-based full-stack web framework**. If you are a beginner in programming, you may have difficulties understanding some concepts or terms. However, it is essential for you to start familiarizing yourself with those new concepts or terms. You can first skim through this section and revisit this chapter later after reading other chapters, which are more tangible.

The following topics are covered in this chapter.

Topic

1. Web Framework and Django
2. Websites vs. Django Web Apps
3. How Django Handles HTTP Request and HTTP Response
4. Django's MVT Framework
5. Django Templates vs. Django APIs

Web Framework and Django



Web Framework

A **web framework** provides a standard way to build and deploy web applications with **coding architecture** and a collection of libraries or tools that allow developers to create web applications more easily and efficiently.

Using a web framework, you can significantly reduce your coding time.

For example, if you want to build user management functionalities in your web applications from scratch, you need to spend long hours building code; however, when using a web framework, you can use pre-made code by simply importing the code for user-management functionalities.

There are different types of web frameworks, from **frontend-focused** and **backend-focused frameworks** to **full-stack frameworks**, which can be used for both frontend and backend.

The scope of functionalities of web frameworks can be very different. Many front-end frameworks can be just called libraries, depending on the definition.

For example, **Bootstrap** is one of the most popular front-end web frameworks. It provides HTML, CSS, and JavaScript-based design templates and quick styling tools (classes) for layout, components, and other design elements. Bootstrap is very useful for quick styling but cannot be used for backend-related functionalities such as user authentication or database management.

On the other hand, backend or full-stack web frameworks, including Django, can provide user management, database management, or security management functionalities.

Django

Django is a Python-based **full-stack web framework**. It is considered a full-stack web framework that provides front and backend functionalities. (Recently, Django has been becoming increasingly popular for backend functionalities, providing APIs to frontend frameworks such as **React.js**. We'll explain this later.)

There are several benefits of using Django for web applications.

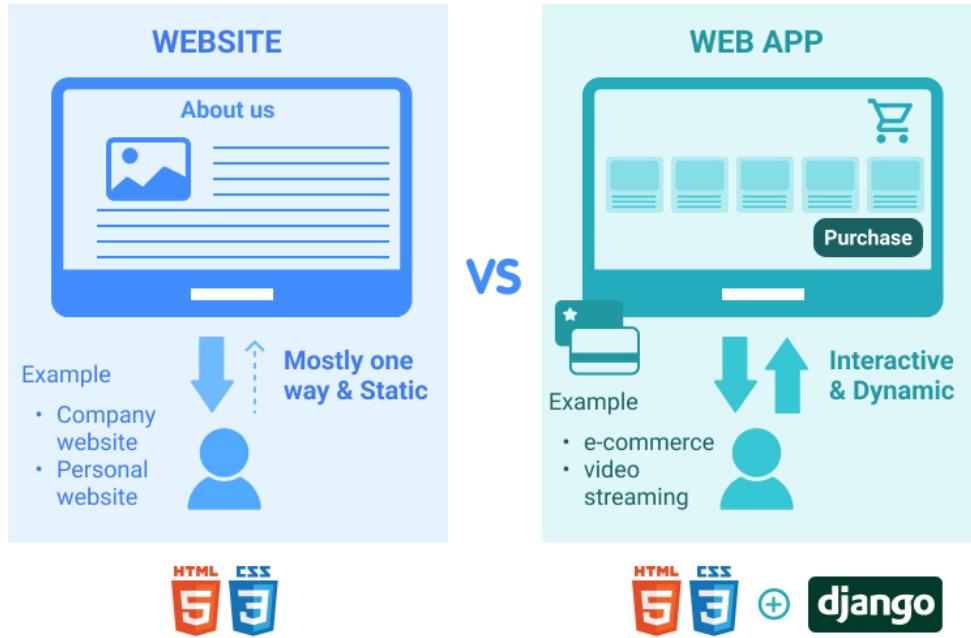
Here are some examples of benefits:

- You'll be able to **minimize your coding time**. Following the Django framework, you can make a simple app within a day.
- You'll be able to access a **built-in admin interface** as soon as you implement a model (database).
- You'll be **released from SQL coding**. Django's Object-Relational Mapping (ORM) system allows developers to interact with database using Python code instead of SQL queries.

- You can **build a secured app** quickly. Django includes many security features, such as protection against cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.
 - You can **scale up your app quickly**. Django is designed to be scalable, meaning it can handle large amounts of traffic and data.
-

If you are a beginner, you may need help understanding what we are discussing, but you'll gradually understand these benefits through this course.

Websites vs. Django Web Apps



To understand what Django can do, comparing websites and web applications may be helpful.

Differences between websites and web applications

There is no strict demarcation, but some critical differences exist between websites and web applications.

Static vs dynamic in user interactions:

- Websites are more static.
- Web applications are more dynamic.

User data handling:

- Websites handle limited user data.
- Web applications store user data in their database and utilize it to provide services.

For example, company websites or personal websites are more static and usually don't handle user data.

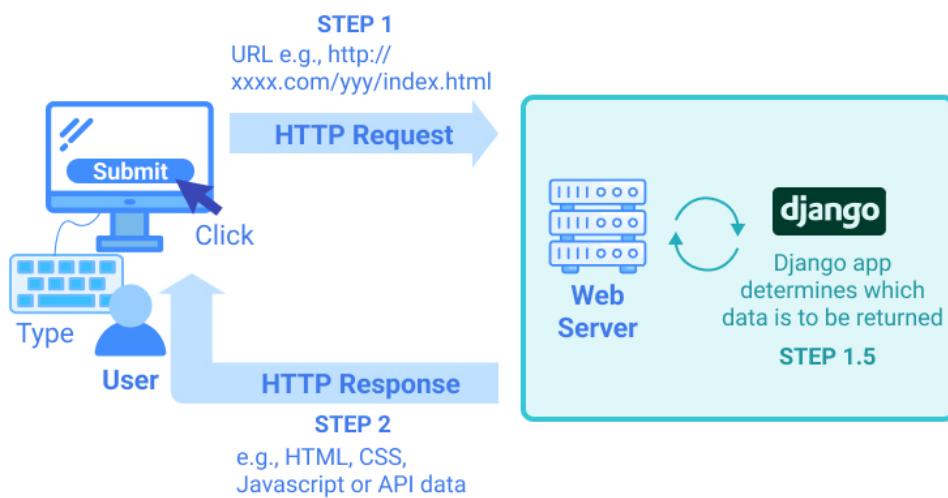
On the other hand, e-commerce like Amazon or video streaming like Netflix provides more dynamic interactions with users utilizing their user data.

Website coding vs. web application coding

Typically, websites are developed with front-end coding such as HTML and CSS (with some JavaScript coding for dynamic visual effects).

Web applications typically require more effort in backend coding to add dynamic functionalities to web applications, including user data handling. Here, Django comes into play. Django is one of the most popular web frameworks to provide dynamic functionalities. For example, Instagram, Spotify, Pinterest, and many other companies are using Django in their services.

How Django Handles HTTP Request and HTTP Response



To describe Django's role in web applications, it is crucial to know the basics of **HTTP requests** and **HTTP responses**. HTTP is a communication protocol used for websites or web applications.

The basic flow of browsing websites

The basic flow of browsing websites (without dynamic components) can be described in two simple steps.

Step 1: When a user clicks on a link or button in a web browser, the browser sends an HTTP request (URL) to a web server located at the address specified by the URL.

Step 2: The web server sends an HTTP response (requested resources such as HTML, CSS, and image files) to the browser, and the browser renders the data on the display.

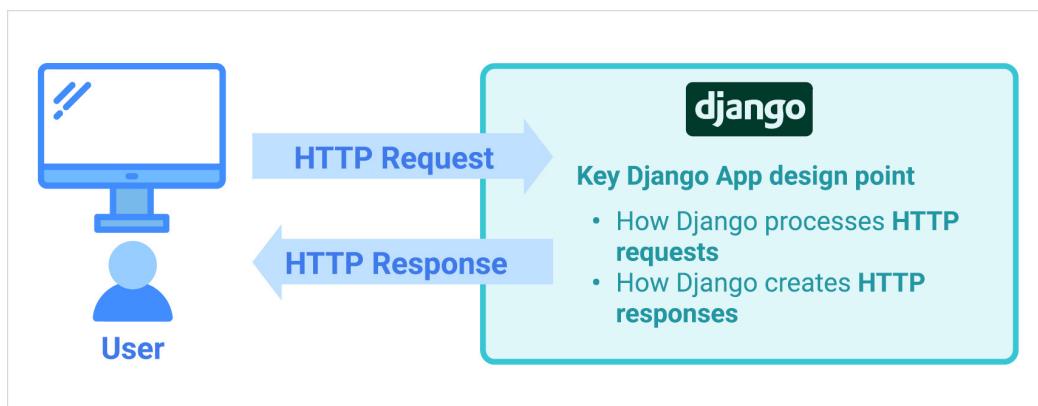
How is a Django application involved in the process?

If a user accesses Django-based web applications, there will be one more step between HTTP request and HTTP response.

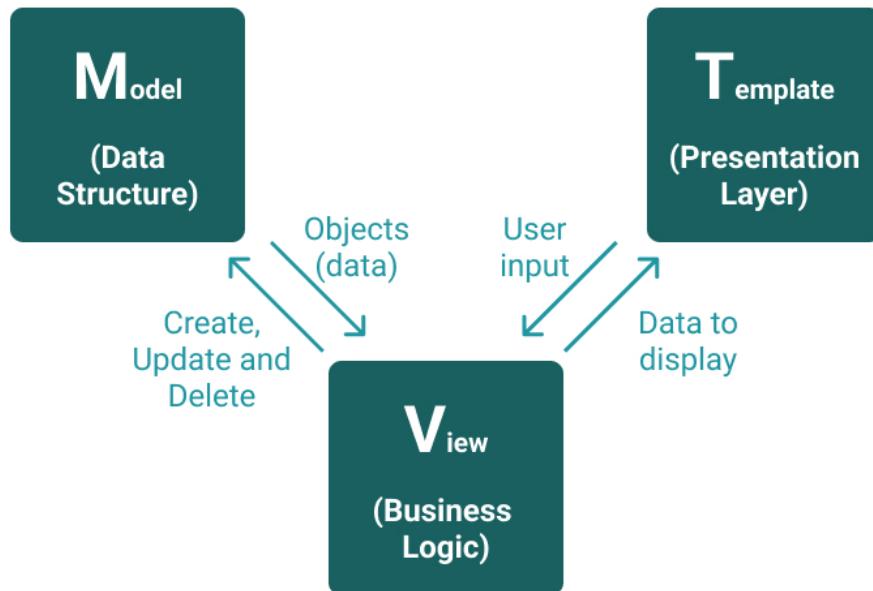
Step 1.5 (between Step 1 and 2):

Once the web server receives an HTTP request, the web server passes the HTTP request to the Django application, and the Django application processes the request. Depending on the HTTP request, Django generates a tailored HTTP response and sends it back to the web server.

Thus, designing a Django web application is about designing processes when the Django application receives HTTP requests and creates HTTP responses.



Django's MVT Framework



Django uses the **MVT (Model-View-Template)** architecture (or, sometimes called, **MTV** architecture). The MVT architecture is similar to the well-known **MVC (Model-View-Control)** architecture at a high level; however, they are different concepts in detail.

Using the MVT framework, Django separates an application design into three components. This separation makes it easier to develop and maintain the code in a more organized and structured way.

1. **Model:** Handles data structure through interacting with a database. The code for Model is written in Python and is usually done in the *models.py* file.

2. **View:** Handles business logic. For example, filtering data from Model for a specific page. The code for View is written in Python and is usually done in the `views.py` file.
3. **Template:** Handles the presentation layer to display the data provided by View in the browser. Files used for the Template are written in HTML format. You can customize filestructure the way you handle typical HTML documents, but Django provides an additional language called **DTL (Django Template Language)** used to bridge HTML and Python code.

A basic flow in the Django Basic Architecture

Let us explain the architecture with the HTTP request and response flow with key coding files to give you a more concrete idea. As the details will be explained in the following chapters, you can focus on more high-level concepts in this section.

In the actual implementation, you also need some other components of Django design architecture.

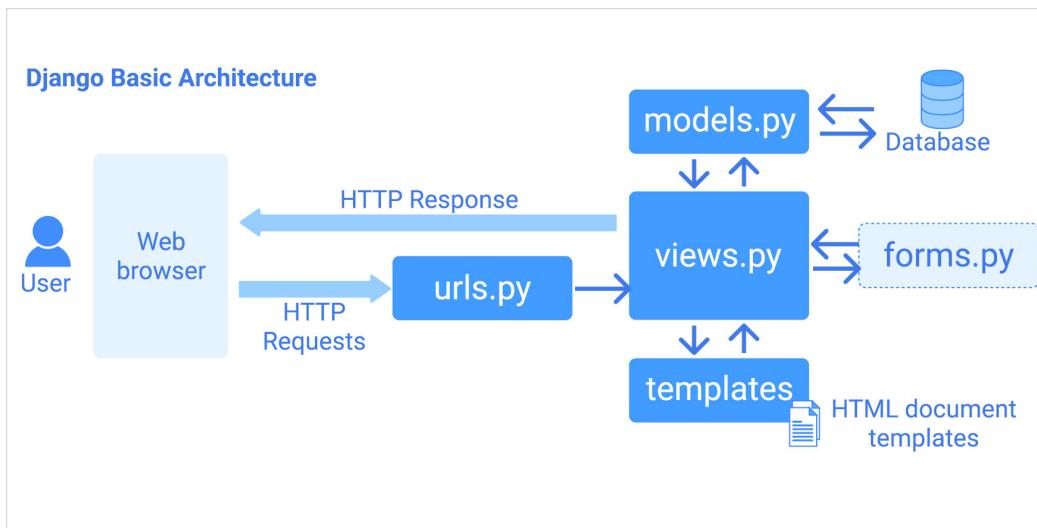
URL dispatcher: Handles URL requests to pass the request to View. It is written in the `urls.py` file.

Form: Handles user data input validations. This is optional as Django's built-in View inherits basic form functionalities.

Here are the basic steps when Django handles an HTTP request and returns an HTTP response:

1. When the **URL dispatcher** (`urls.py`) receives an HTTP request from a browser (through a web server), it defines which View (written in `view.py`) should be called.
2. The **View** called by the URL dispatcher processes the HTTP request, working with Model, Template, and Form to create an HTTP Response.

3. The **Model** provides data from a database based on the View's instruction.
4. The **Template** converts the data from View in the HTML format.
5. When necessary, **Form** validates user data input.
6. Once the HTTP response is ready, the data is sent back to the web browser.



In the next chapter, we'll explain each component (except for forms.py) in detail using this diagram.

Note: MVC architecture

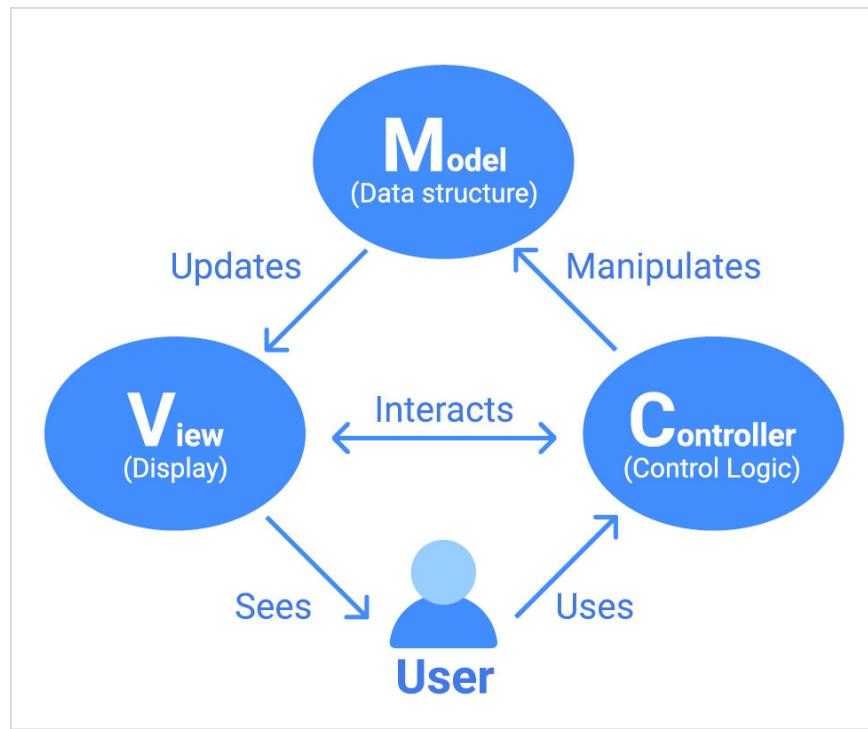
MVC architecture is a software architectural design pattern that is widely used in GUI-based applications, including web applications.

MVC architecture has three components:

Model: Handles data structure and logic through interacting with a database.

View: Handles visual representations by displaying the data to users.

Controller: Handles user interactions and manipulates Model while interacting with View.

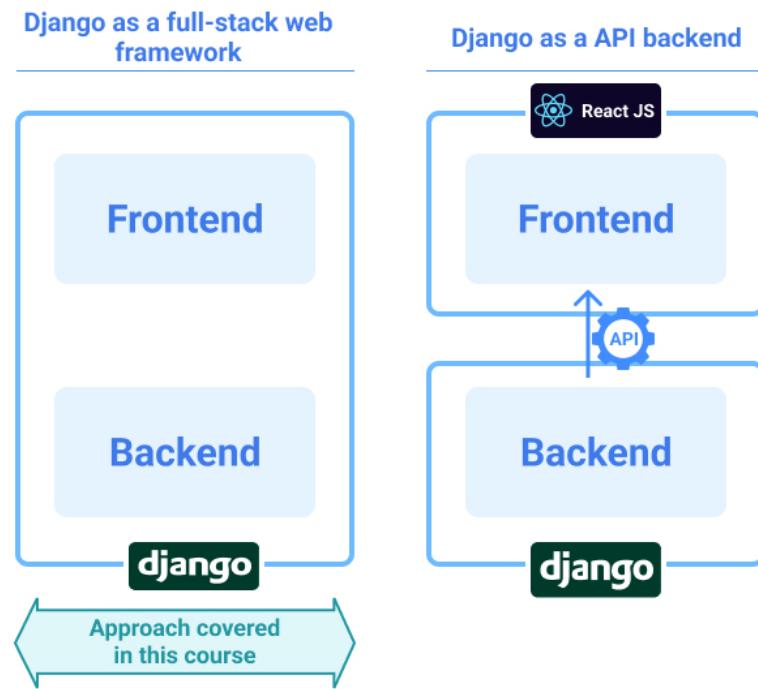


The difference between MVC and MTV

The official Django documentation states that it could use the term **Templates for Views in MVC** and the term **Views for Controller in MVC**, and Django itself takes care of the Controller part.

[Django documentation reference: FAQ: General.](#)

Django Templates vs. Django APIs



There are two approaches for front-end coding when using Django in web application development.

One uses Django as a **full-stack web framework** with Django templates. The other uses Django as **an API backend** while using another frontend framework, such as React.js.

Recently, the API approach has become more popular; however, we'll explain how Django works using the first approach in this course, as that approach gives you a better understanding of Django.

Chapter 02

Django Quick Start Guide

This chapter covers key concepts used in Django, a Python-based full-stack web framework. If you are a beginner in programming, you may have difficulties understanding some concepts or terms. However, it is essential for you to start familiarizing yourself with those new concepts or terms. You can first skim through this section and revisit this chapter later after reading other chapters, which are more tangible.

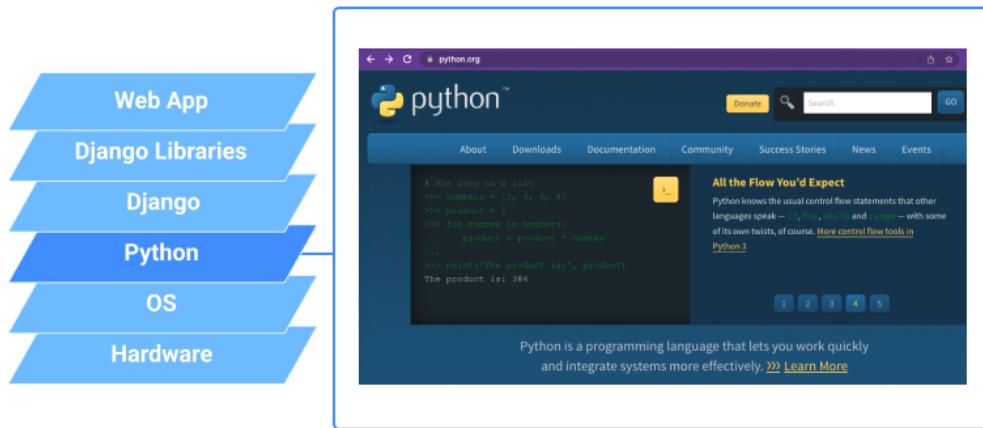
The following topics are covered in this chapter.

Topic

- 1. Install Python**
- 2. Install Visual Studio Code**
- 3. Create Project Directory**
- 4. Set Up Virtual Environment**
- 5. Install Django**
- 6. Start Django Project**
- 7. Run Server**
- 8. Database Migration**

- 9. URL dispatcher – urls.py**
- 10. Create Superuser and Log In to Django Admin**
- 11. Start App**
- 12. Create HTML Templates**
- 13. Create Views**
- 14. Project vs. App**
- 15. Template with Model Relations**

Install Python



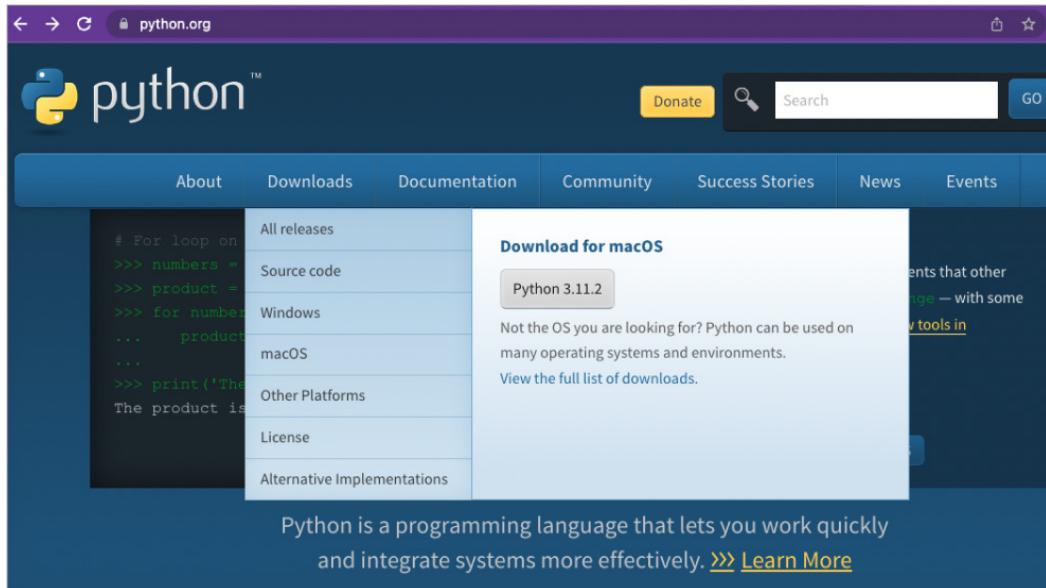
Django is a Python-based web framework, so you must have Python on your computer. Depending on your computer's OS, Python may be already pre-installed; however, Python has several versions. It is better to use the latest version unless you have a specific reason to use older versions.

For Mac and Windows OS

Download and install Python from the official site

You can download the latest Python from the [Python's official site](#).

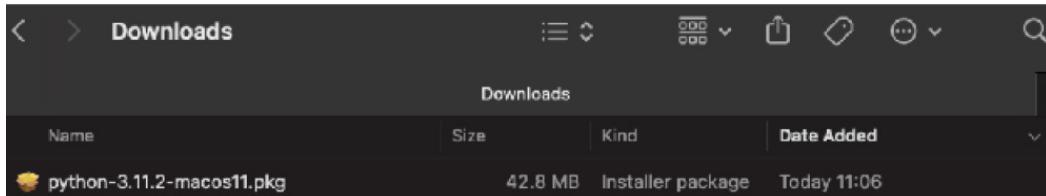
1. Go to the official Python site and select your OS under Downloads.



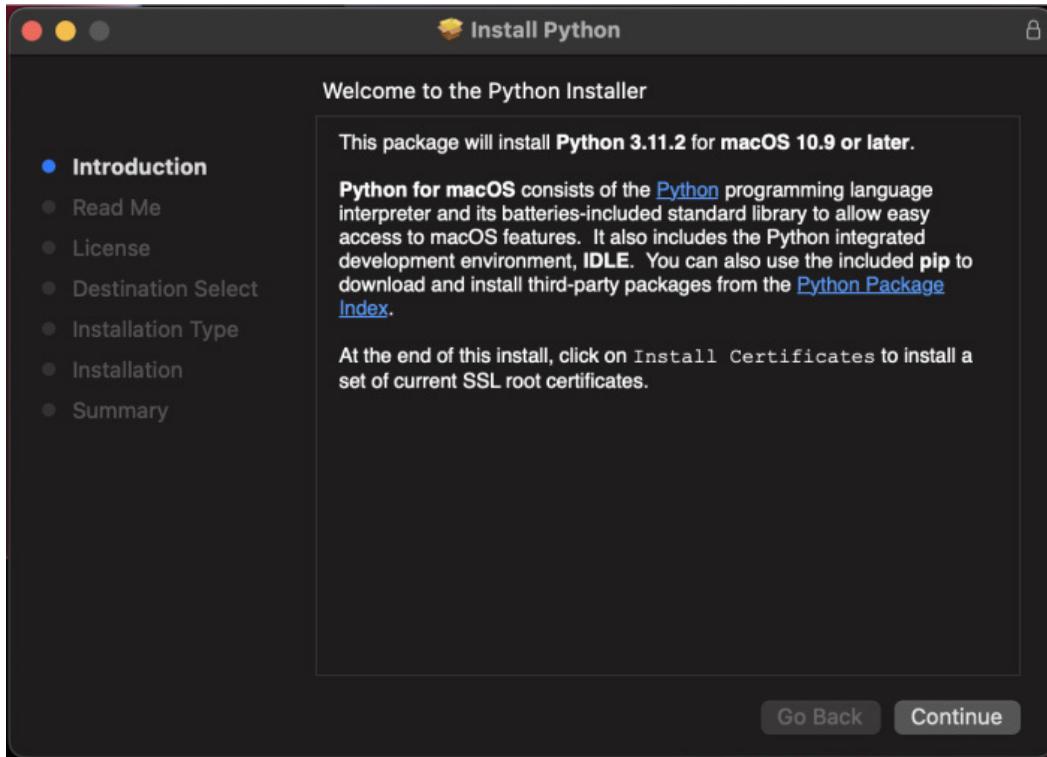
2. Install Python and check the installed version

For Mac

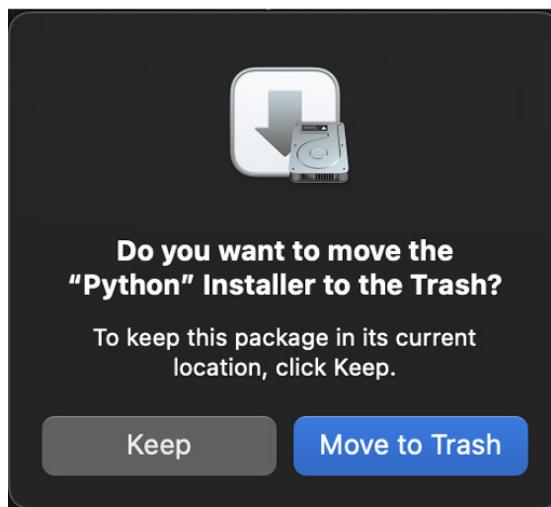
Downloads pkg file



Open the pkg file and follow the instructions



Move the installer to Trash



Check the installed version

Run the `python3 --version` command. As the latest version of Python is version 3, you need to type `python3`.

Command Line - INPUT

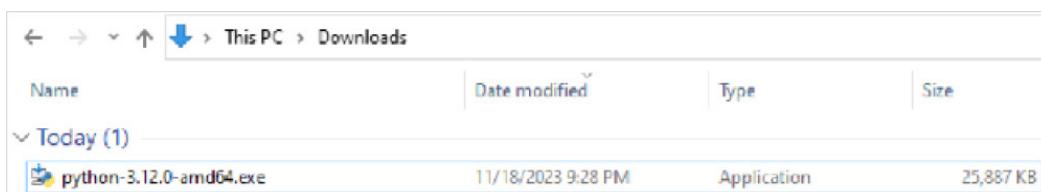
```
$ python3 --version
```

Command Line - RESPONSE

```
Python 3.11.2
```

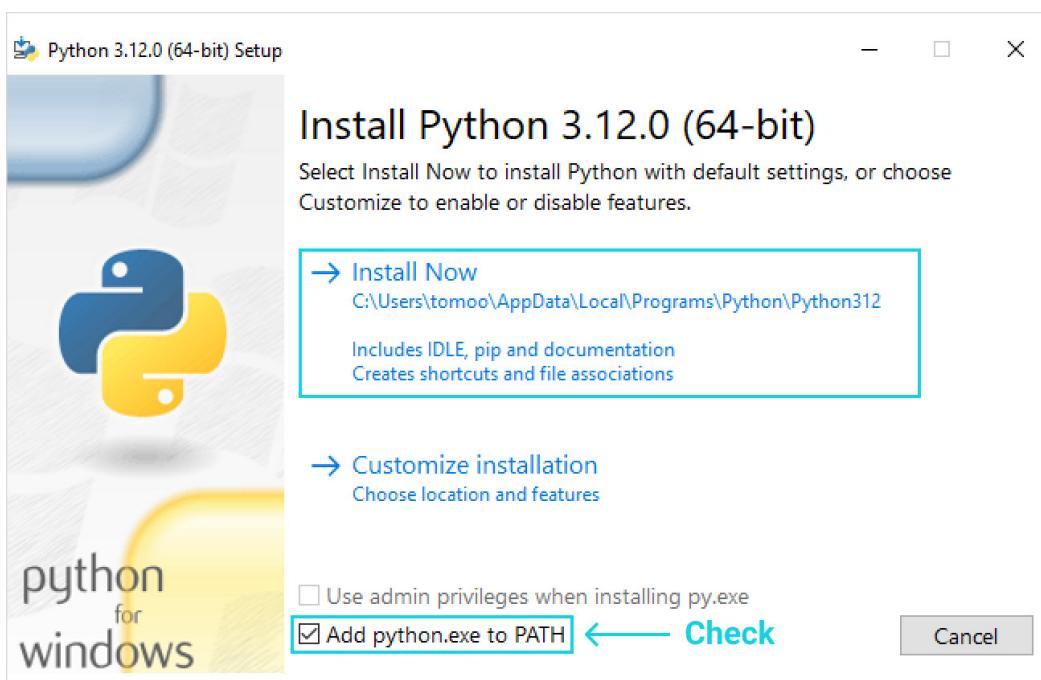
For Windows

Download the exe file

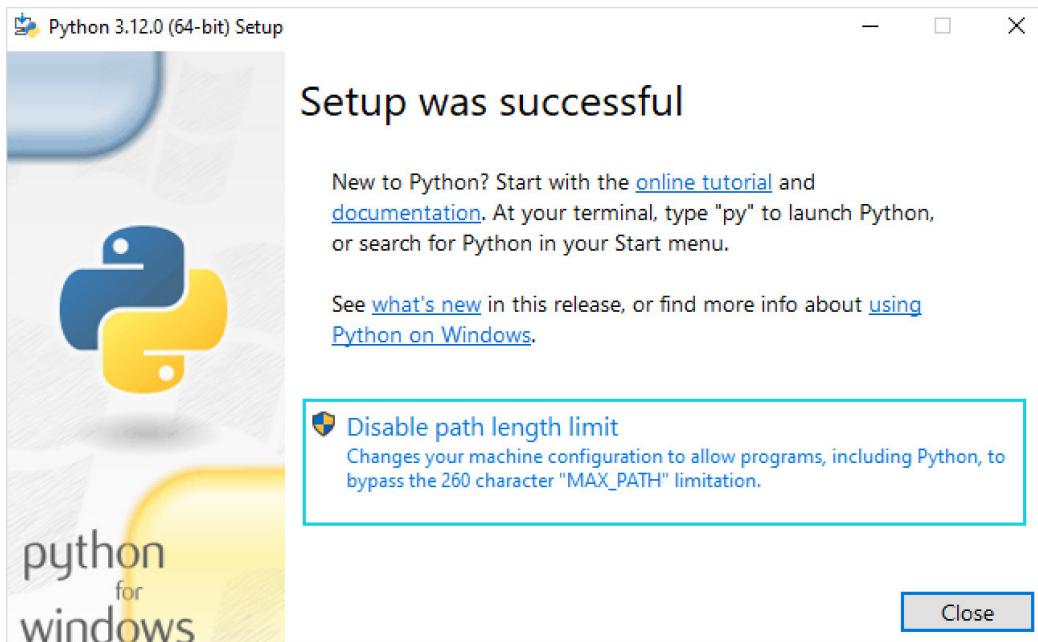


Install Python

Make sure you check “**Add python.exe to PATH**”. If you don’t check this, your command line may not recognize that Python is installed.



Note: If you may use a long file path, click **Disable path length limit** before closing the popup.



Check the installed version

Run the `python --version` command. For Windows, you don't need to use `python3`.

Command Line - INPUT

```
$ python --version
```

Command Line - RESPONSE

```
Python 3.11.2
```

For Linux OS

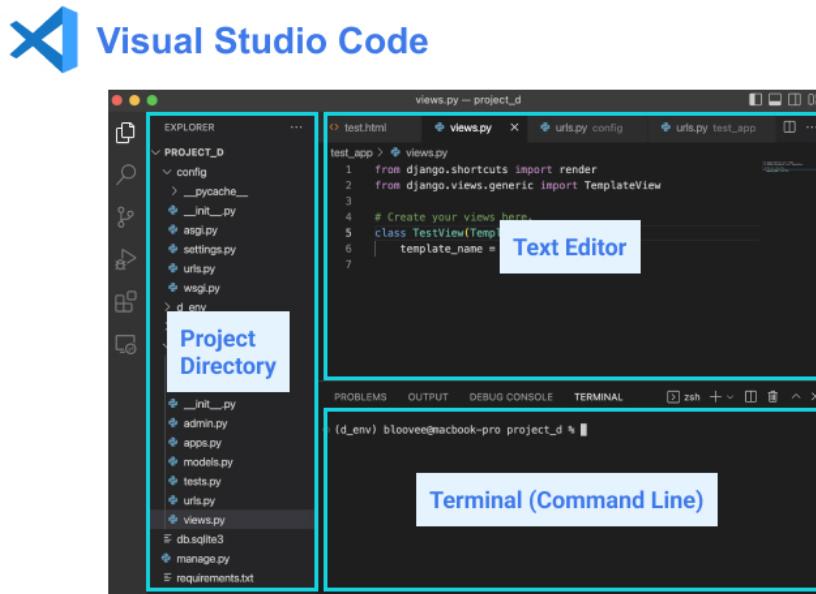
Install Python using a package manager in the command line

If you are using the CUI environment, you need to use a package manager command. For example, run the following commands for ubuntu Linux OS.

Command Line - INPUT

```
$ sudo apt update  
$ sudo apt install python3
```

Install Visual Studio Code



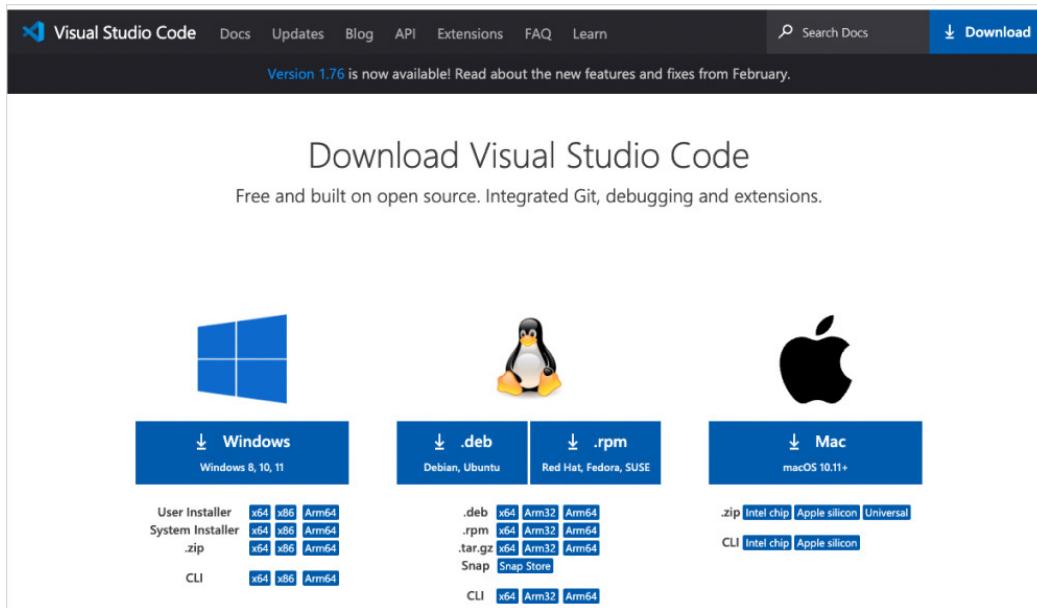
For Django app coding, you can use a simple text editor and a command line; however, using an advanced code editor tool such as **Visual Studio Code (VS Code)** can improve your coding productivity. VS Code is a free software provided by Microsoft. It is one of the most popular code editors among code developers. In this course, we use VS Code as a default editor.

Install and prepare VS Code on your desktop

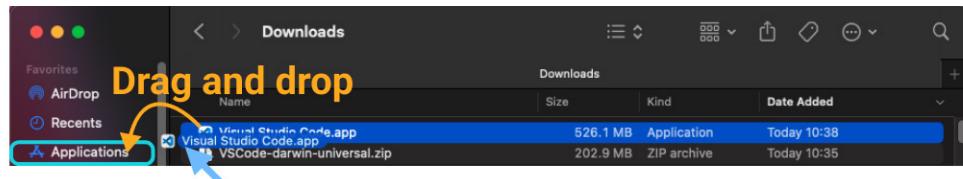
VS Code is available on the [VS Code's official site](#).

Here are the steps to install and prepare the VS Code app on your desktop with [Mac OS examples](#).

1. Go to the official site and select your OS.



2. Go to the downloads folder. Drag and drop the Visual Studio Code.app icon onto the Application folder.



3. Open Launchpad from the Dock. You can find the Visual Studio Code app.



4. To create an icon in the Dock, drag and drop the Visual Studio Code app onto the Dock from Launchpad.



5. You can open the VS Code from the Dock now.



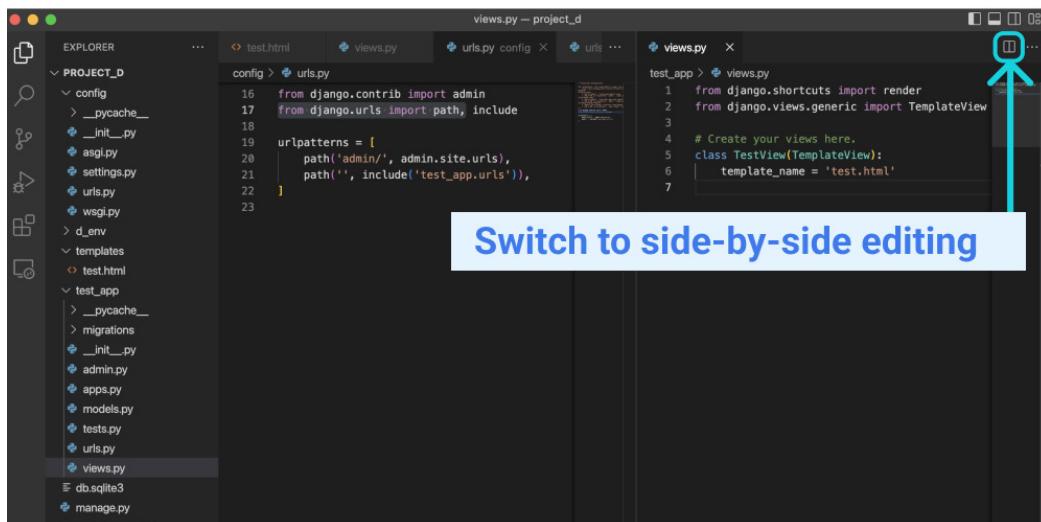
Key features of VS Code

There are many features of VS Code. As a beginner, you'll find it to be a valuable tool because of its integrated user interface.

For example, you can see the following views on the same screen.

- 1. Your project directory structure.**
- 2. Text file contents with multiple tabs.**
- 3. Terminal (command line) to run commands.**

Using VS Code, you can also open two active files side-by-side. This is helpful when you want to compare the code in a file with the code in another file. Click on the top right icon to split the active editor into two.

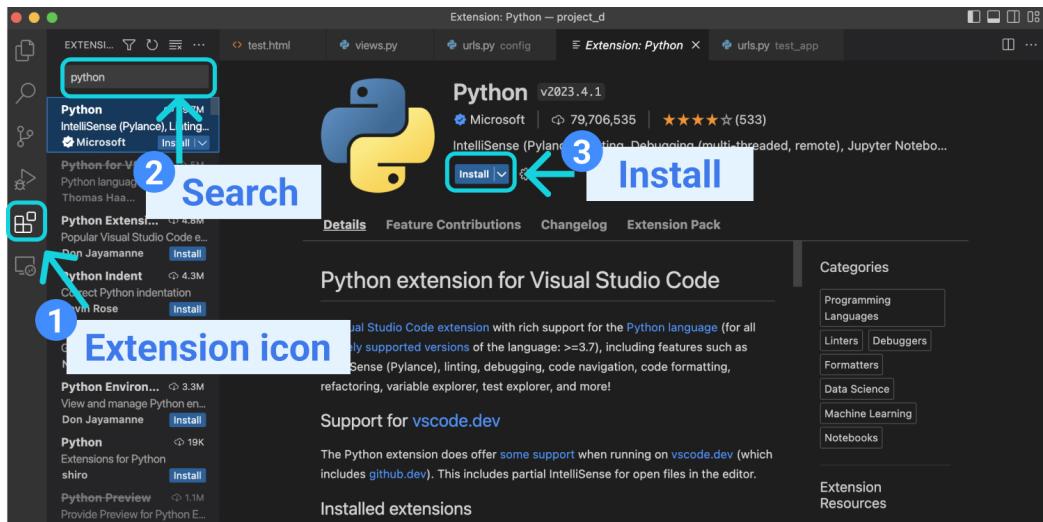


Extensions

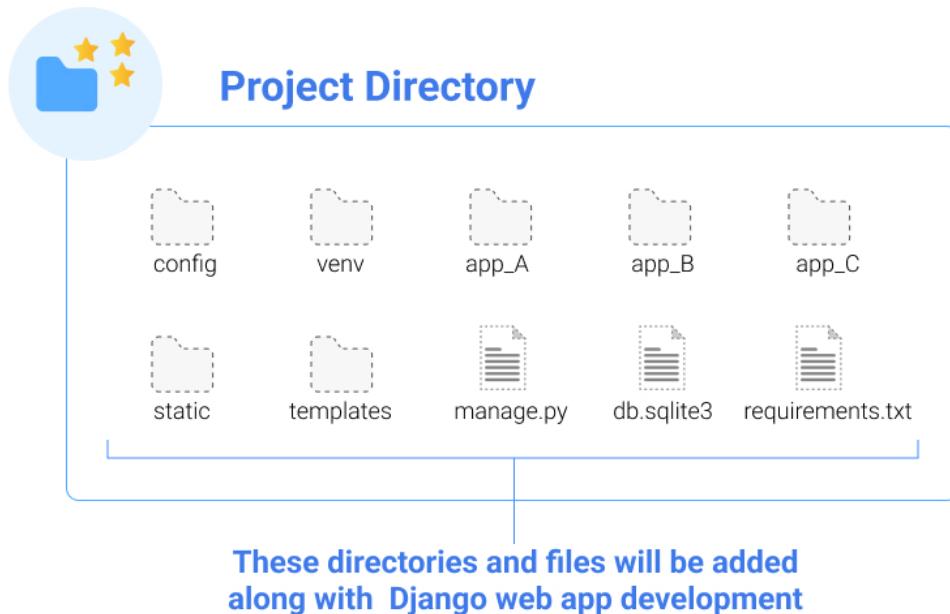
VS Code has many extensions that provide helpful features, such as a **Remote Development** extension for SSH connection and a **Docker** extension to manage Docker assets. Python extension is a popular extension used for Python code debugging, code formatting, and several other purposes.

With the three steps below, you can install the Python extension.

1. Click on the **Extension** icon on the left sidebar.
2. Type “**python**” in the search bar and search extensions.
3. Select the **Python** extension and press the Install button.



Create Project Directory

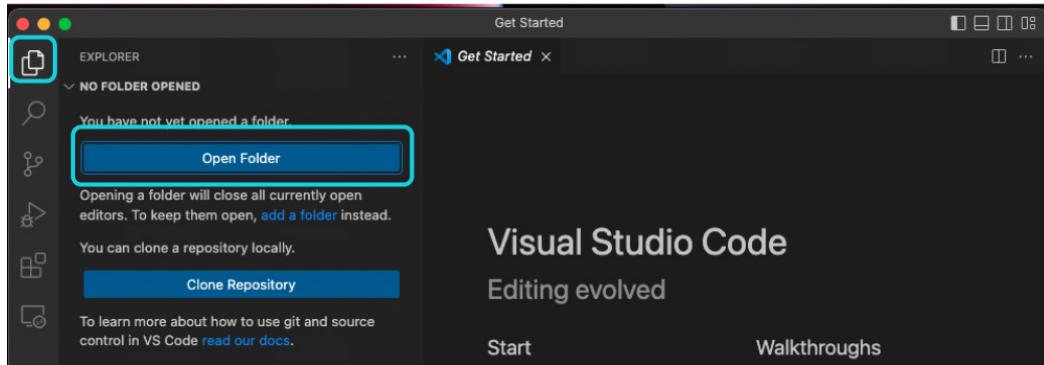


Creating a project directory is the first step for starting an app development project. You can create it in your home directory, on your desktop, or in other locations on your computer. Developing a web application using Django will be done in this project directory throughout this course. Many new directories or files will be added to the project directory along with your app development process.

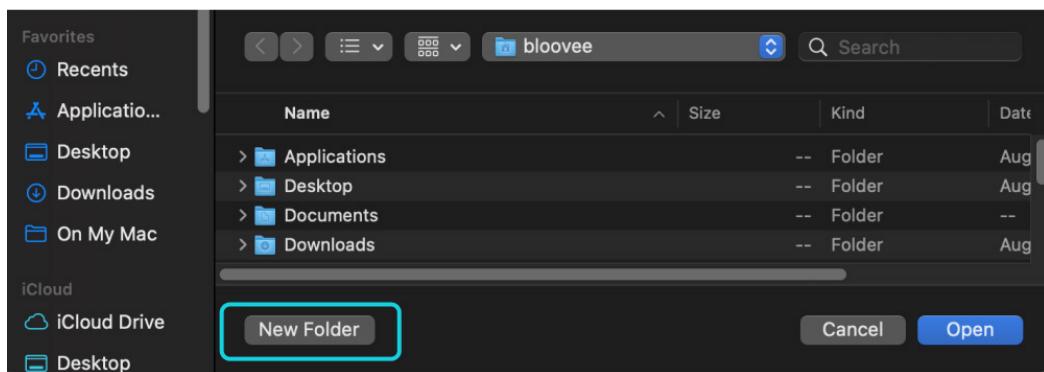
Create a project directory with VS Code

You can use VS Code to create a new project directory. Here is the guide to creating a new project directory with VS Code.

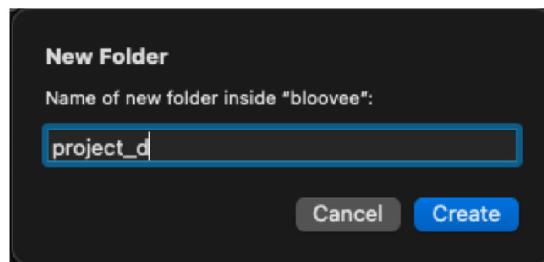
1. Click on the document icon on the top left and press the Open Folder button.



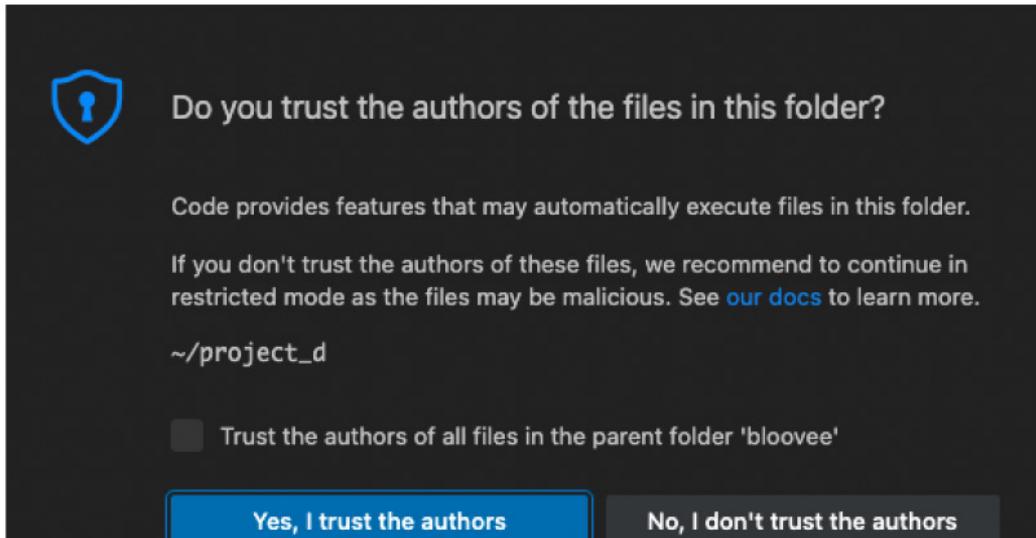
2. Press the New Folder button in the directory selection screen.



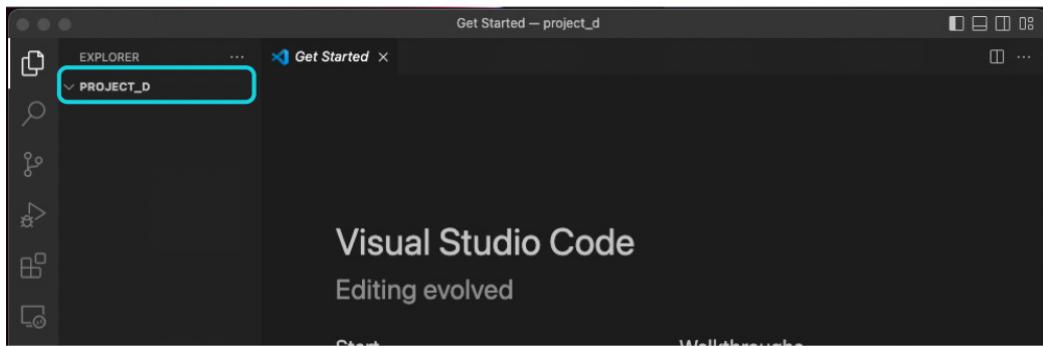
3. Type a project name (e.g., project_d) and press the Create button.



4. When opening the directory that you created, you may be asked if the directory (folder) can be trusted. Select the Yes button to open the directory.

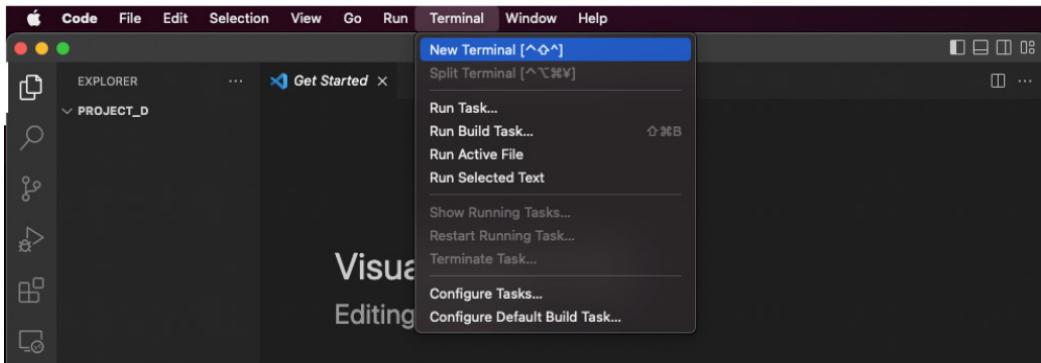


5. You'll see that the project directory has been created like shown below

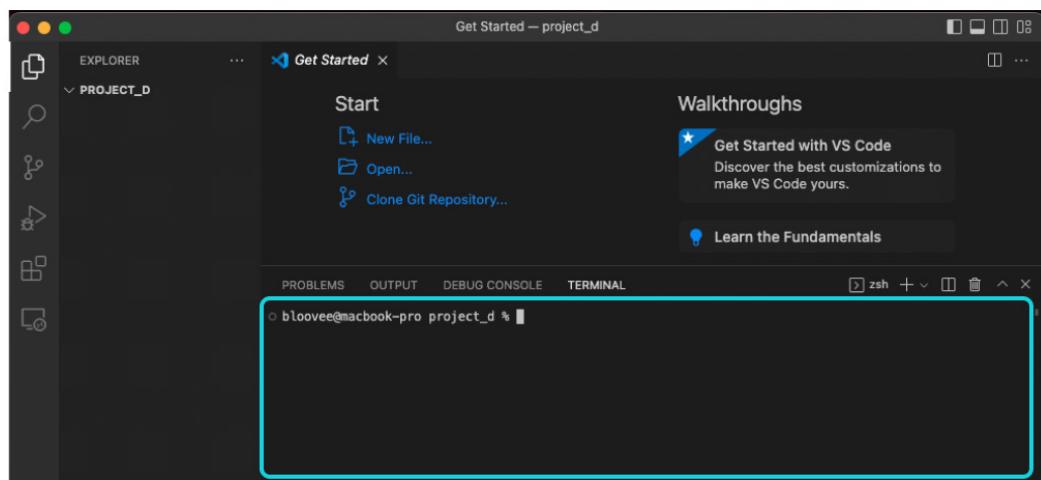


Open Terminal

For the following sections, you'll need to use Terminal. To open the terminal, click on '**Terminal**' in the top menu bar and select '**New Terminal**'.



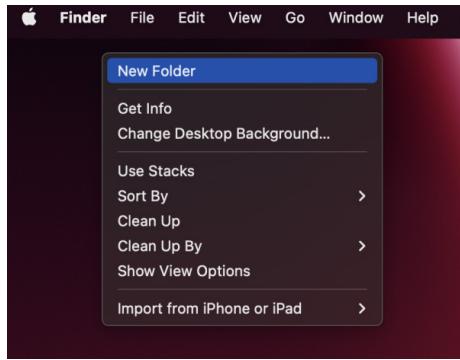
You can see that a new terminal is opened with the project directory as the current working directory.



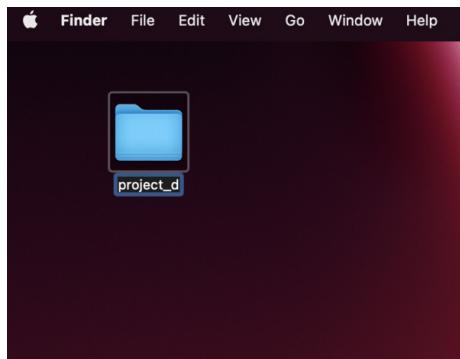
Tips: Start VS Code with drag and drop

You can also create a new project directory with your Mac OS or Windows and open it with VS Code. Here are the steps for Mac OS.

1. Create a new folder by right-clicking on your desktop.



2. Change the folder name.

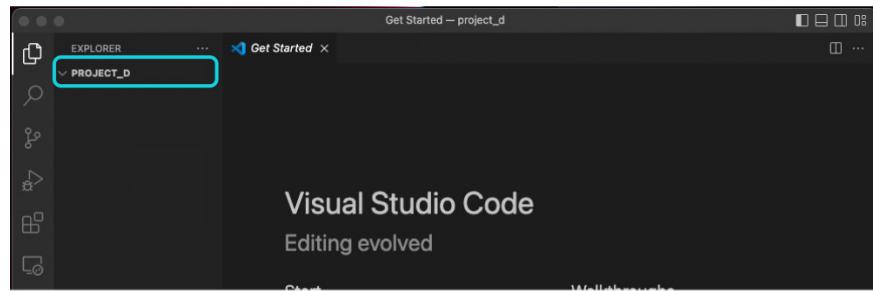


3. Drag and drop the folder onto the VS Code icon.

- **For Mac:** use the VS Code icon in the Dock.
- **For Windows:** use the VS Code icon on the desktop.



4. You can see that the project directory is opened with VS Code

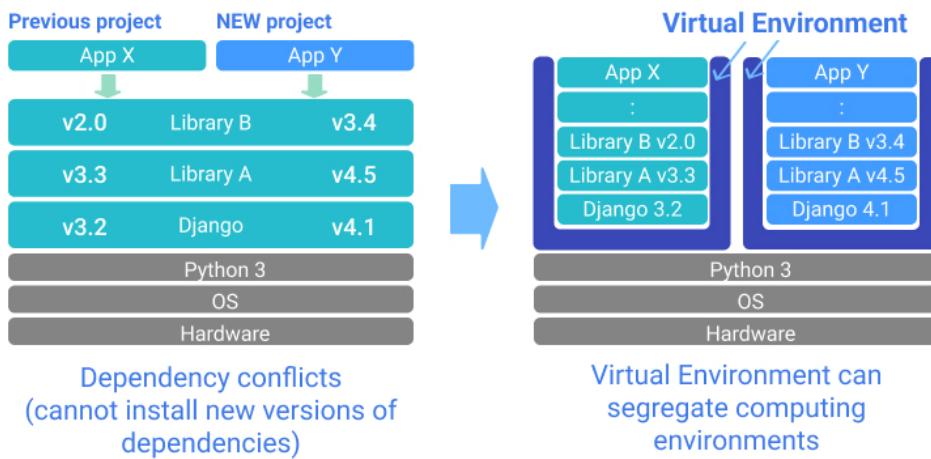


Set Up Virtual Environment

[xx] argument

```
python3 -m venv [ven name]
```

```
source [venv name]/bin/activate
```



When you develop or run an application, the application will utilize a set of other software programs which are interlinked with each other (called "**dependencies**").

For example, your computer runs on a particular OS (e.g., Mac OS X Yosemite). And on the OS, particular versions of the programming language (e.g., Python 3.8), web framework (e.g., Django 3.2), and/or many libraries are installed.

When you develop a Django application, you need to pay attention to the dependencies. Your application can adequately run on the dependencies you configured; however, it may not work correctly on another computer as versions of libraries, python, or Django itself may differ across computers. The environment dependent on each computer is called the **computing environment**.

Virtual Environment

A **virtual Environment** is a concept or tool used to create a specific environment for an application by segregating it from other computer resources. There are several ways to create a virtual environment. Using **virtual machine** software is one choice. The recent popular approach is using a **Docker container**. But these approaches take time to set up. Python has a feature to create a virtual environment by running a simple command.

venv

`venv` is a command to create a virtual environment in your project directory with a few steps.

Step 1: Create a virtual environment directory

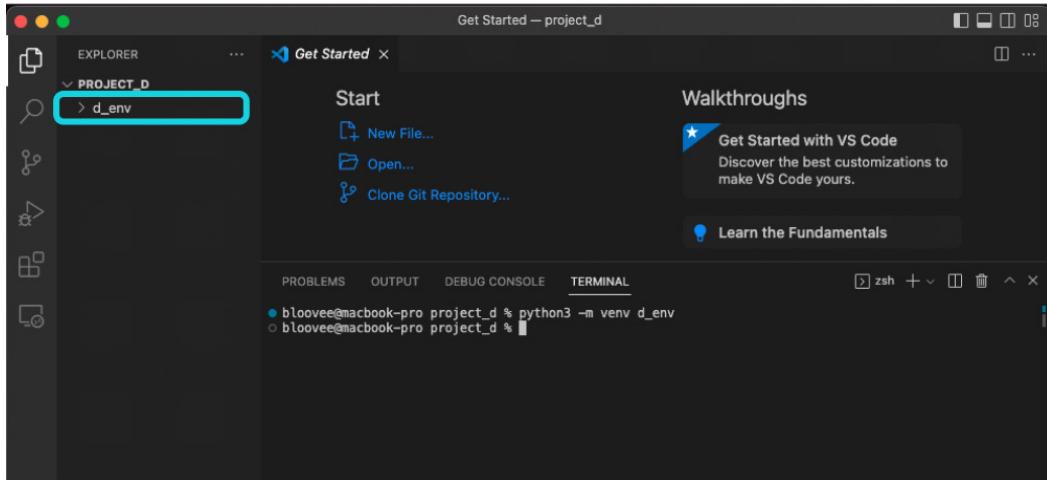
In the project directory, run the `venv` command with a virtual environment name you want to create. The example below uses `d_env` as the virtual environment name.

Command Line - INPUT

```
project_d % | python3 -m venv d_env
```

Note: For Windows, use `python` instead of `python3`.

When you run the command, the `d_env` directory is created directly under the project. The dependency information will be saved under this directory throughout your code development.



Step 2: Activate the virtual environment

To create a virtual environment in your project directory, you need to activate your virtual environment by running the `source` command. For Mac, the activate file is stored under the `bin` directory under the virtual environment directory you created in the previous step.

Run the command below to activate the virtual environment named `d_env`.

Command Line - INPUT

```
project_d % | source d_env/bin/activate
```

When you run the command, there will be a change in the command line interface like the one below. (`d_env`) means the virtual environment named `d_env` is now active.

Command Line - INPUT

```
(d_ev) project_d % |
```

When you install Django, you need to make sure that your virtual environment is active to manage your new app's dependencies properly.

For Windows (Powershell)

For Windows, the command used to activate the virtual environment is different from the one on Mac.

When activating the virtual environment for the first time, you must first run the command below.

Command Line - INPUT

```
project_d % | Set-ExecutionPolicy -ExecutionPolicy  
RemoteSigned -Scope CurrentUser
```

Then, run the command below to activate the virtual environment.

Command Line - INPUT

```
project_d % | .\d_env\Scripts\Activate.ps1
```

Step 3: Deactivate

You can run the `deactivate` command when you want to return to the normal mode.

Command Line - INPUT

```
project_d % | deactivate
```

After running the command, you'll see that the command-line interface goes back to normal.

Command Line - INPUT

```
project_d % |
```

Note: venv directory location

The `venv` directory location should not be changed. If you move the directory, python will refer to the original absolute path, and you won't be able to run the Python command properly.

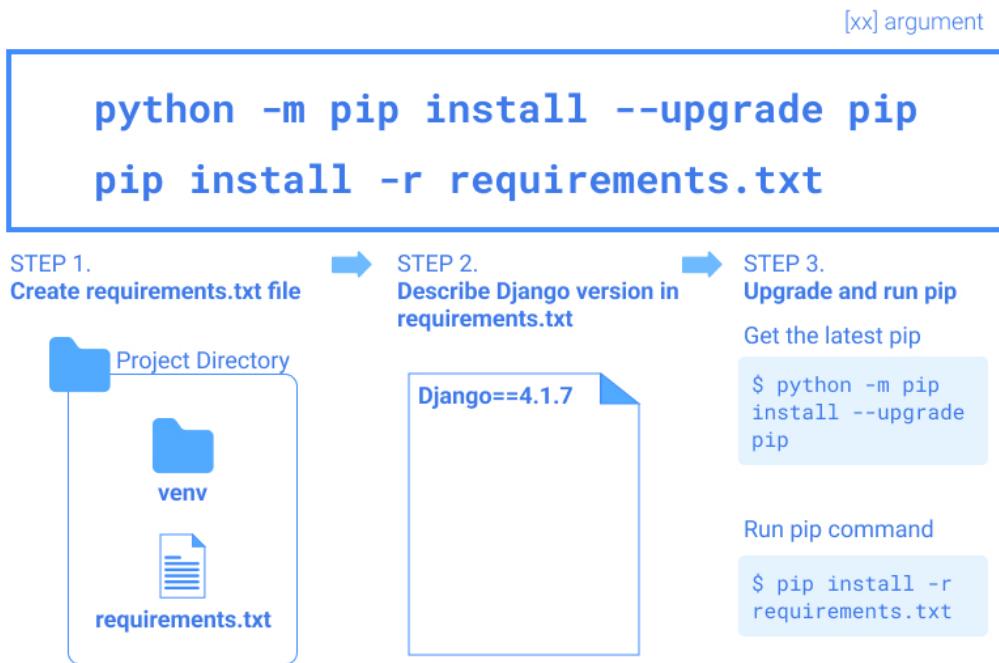
Thus, you must also be careful when moving your project directory. If you move your project directory,

your *venv* directory will also move.

If you want to change the location of the project directory, delete the existing *venv* directory and create a new *venv* directory under a new path. If you are using the same *requirements.txt*, you should be able to recover the virtual environment fully.

Also, you need to make sure that you don't edit files and directories under the *venv* directory during your coding.

Install Django



The Django installation guide is available on the [Django official site](#).

You can install Django by simply running the `pip` command. Pip is **Python Package Manager**. The best practice for installing Django is using the *requirements.txt file*. As you will likely install other libraries during the Django app development process, it is better to record the library information in one place. You can add libraries to install for your app in the *requirements.txt* file. Here are the critical steps for Django installation.

Step 1. Create a requirements.txt file

Create a text file named '`requirements.txt`' directly under the project directory. Click the file icon on the left sidebar of the VS Code to create a new file.

Step 2. Add Django with the version name in the requirements.txt file

Check the latest version on the download page on the [Django official site](#).

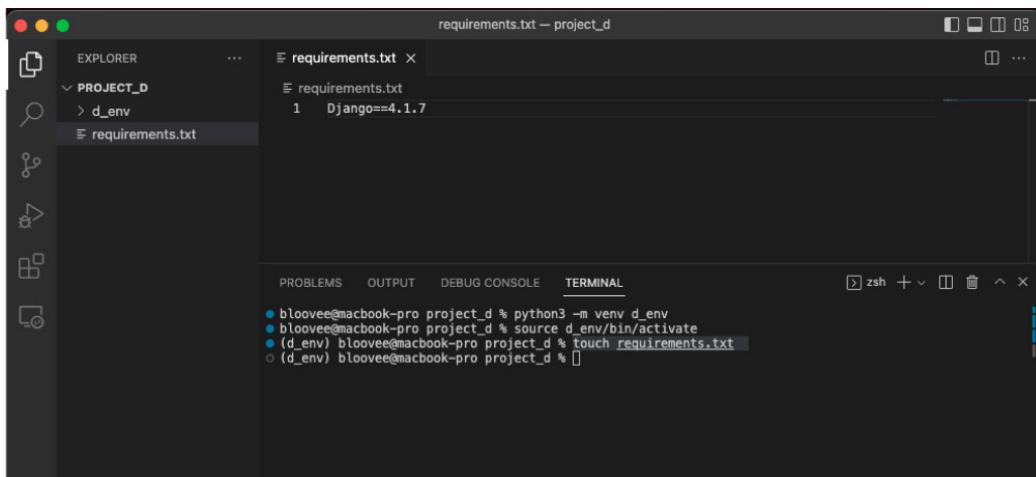
You can install Django without specifying a version; however, it is better to specify it to manage dependencies properly.

The example below is for installing Django version 4.1.7. Do not forget to save the file when you finish editing it.

requirements.txt

```
Django==4.1.7
```

This is a screenshot of the VS Code for your reference.



Step 3. Update and run the pip command

To install Django, you can use the pip command. To make sure the pip command is the latest one, upgrade it by running the command below.

Command Line - INPUT

```
(d_env) project_d | python -m pip install --upgrade pip
```

You'll see the pip version when it is successfully upgraded.

Command Line - RESPONSE

```
: Successfully installed pip-23.0.1
```

After upgrading the `pip`, run the `pip` command. Ensure that the virtual environment is active when running the `pip` command. If your command line prompt starts with `(virtual environment name)`, the virtual environment is active. Here is a command example.

Command Line - INPUT

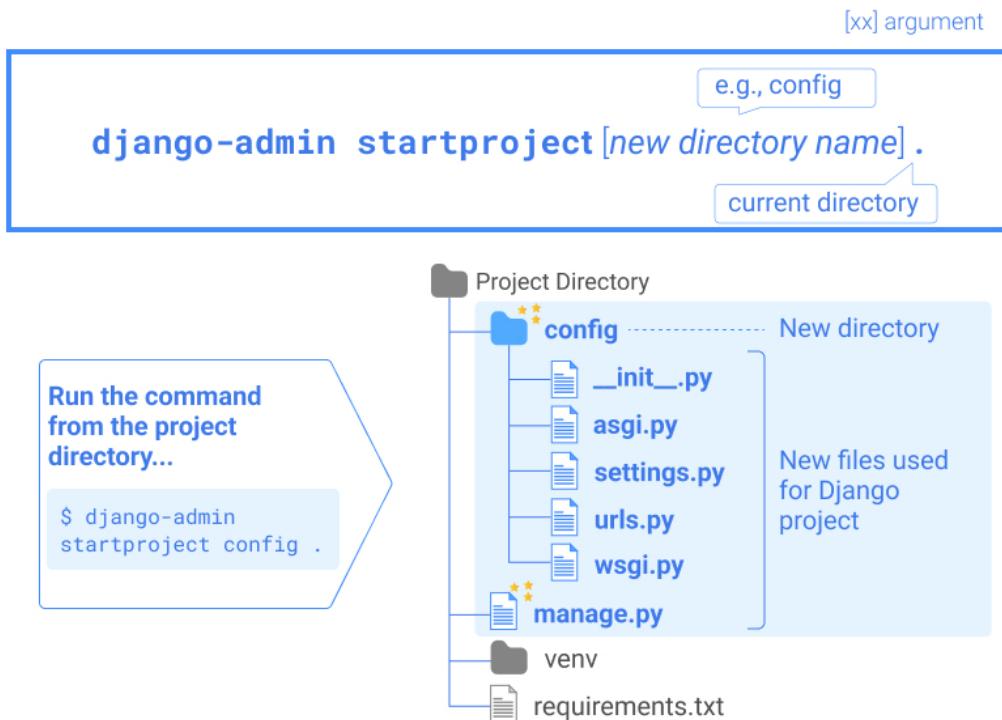
```
(d_env) project_d | pip install -r requirements.txt
```

You'll see that the specific version of Django is installed along with some required libraries.

Command Line - RESPONSE

```
: Successfully installed Django-4.1.7 asgiref-3.6.0 sqlparse-0.4.3
```

Start Django Project



To develop a new app with Django, you need to run the `django-admin startproject` command. The command creates key files used for a Django project.

The django-admin startproject command

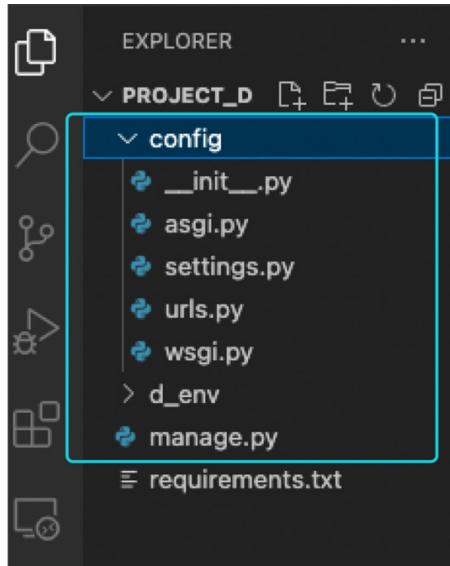
You can specify the name of the new directory when you run this command. As the files to be created under the new directory are used mostly for project configuration, the commonly used directory name is 'config'. This makes directory management more intuitive later on.

Run the command below from your project directory to start a Django project.

Command Line - INPUT

```
(d_env) project_d % | django-admin startproject config .
```

You'll see that several files are created like shown below.



Django key files created by the django-admin startproject command

When you run the `django-admin startproject` command, several files are created. Here are simple explanations of each file.

manage.py (Important)

This file is created directly under the project directory, while other files are created under a new directory. The `manage.py` file is used to run several Django commands. You'll see how to use this file in the following sections.

__init__.py (Important)

This particular file for Python programming is used to mark a directory as a Python package. Often, there is no content. In regular use, you don't need to touch this file.

asgi.py

This Python module defines the **ASGI** (Asynchronous Server Gateway Interface) application for your Django project. This file is

used when you deploy a Django application. If you are a beginner in Django, you don't need to worry about this file.

settings.py (Very Important)

When you develop a Django app, you'll be frequently touching this file. This file is used to set the Django application configurations. The details of *settings.py* will be explained later in this course.

urls.py (Very Important)

This file functions as a Django **URL dispatcher** (a URL routing system). In the file, you can define URL patterns that are to connect with *views.py*; the latter is used to define a response when there is an HTTP request with the specified URL. How to edit *urls.py* will be explained later.

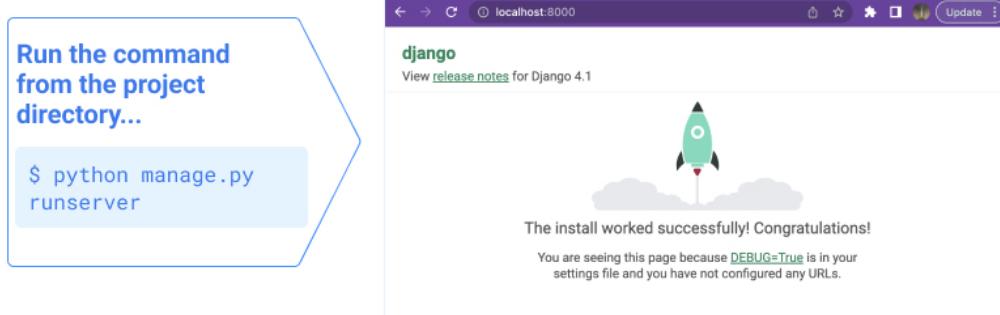
wsgi.py

WSGI (Web Server Gateway Interface) is a standard interface that enables communications between Python-based web applications and web servers such as **Apache** and **Nginx**. *wsgi.py* is a module that implements WSGI. Like *asgi.py*, this file is used when you deploy a Django application. If you are a beginner in Django, you don't need to worry about this file.

Run Server

```
python manage.py runserver
```

Type **localhost:8000** in a web browser



Django provides an easy-to-launch development server used for testing during app development processes. When you run the `runserver` command, your app is deployed in the local environment. The launched server listens on port `8000`. You can access the launched app by typing `localhost:8000` in your web browser.

The runserver command

You can launch your app in your local environment quickly by running the `runserver` command.

We haven't developed any code yet, but you can check how the `runserver` command works by running the command below. The `manage.py` file is used to call the `runserver` command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py runserver
```

When you run the command above, you'll see the message below.

Command Line - RESPONSE

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work
properly until you apply the migrations for app(s): admin, auth,
contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

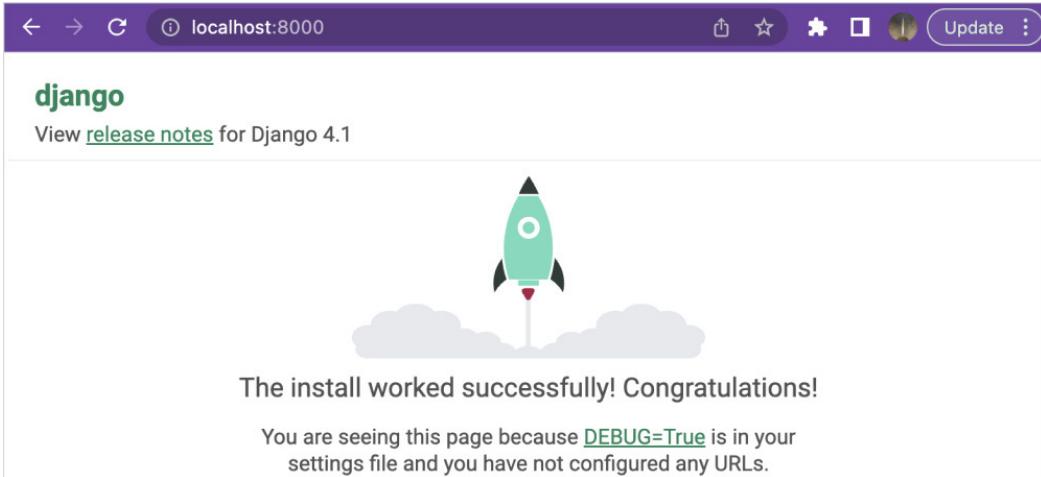
March 13, 2023 - 11:30:31
Django version 4.1.7, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

First, you'll notice that there is an alert saying there are unapplied migrations. This means that pre-designed databases are not implemented yet. For now, you can ignore this message. We'll explain this later.

Accessing the launched Django app

If you look at the latter part of the message in the command line above, you can see "Starting development server at *http://127.0.0.1:8000/*". This means that your Django app is accessible at the IP address and port. **127.0.0.1** is a **loopback IP address** that is also called **localhost**. This address is pointing to your computer.

By typing "*localhost:8000*" in your web browser address bar, you'll see that the Django app is now running.



Quit the server

As the server is running as a foreground process, you need to use keyboard shortcut (**Ctrl** + **C**) to quit the server.

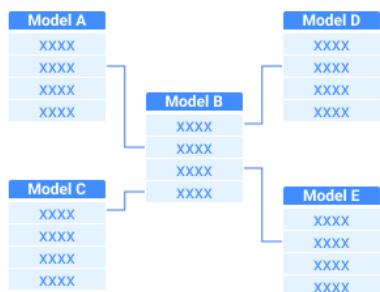
Tips: Python command when venv is activated

When **venv** is activated, you can run a Python command by simply typing '`python`' even on Mac or Linux. No need to type '`python3`' as the command line already recognizes the Python being used is Python 3.

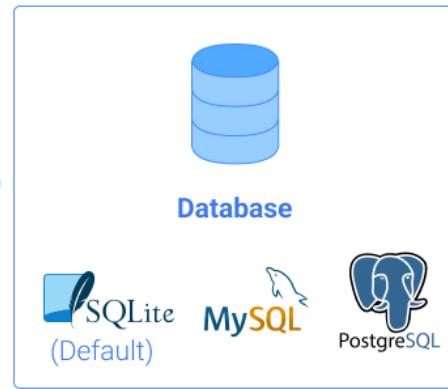
Database Migration

```
python manage.py migrate
```

Database design



migrate



As a default, Django provides basic database designs used for admin and user authentication and some other basic features. As a default, the database design has not yet been reflected in the Django database. This relates to the alert you saw when you ran the `runserver` command in the previous topic. In order to reflect the prepared database design, you need to migrate it into the database.

The `migrate` command

There are three steps to update a database used in Django.

1. Edit `models.py` file: create or modify Models
2. Run the `makemigration` command: create a migration file based on the Models

3. Run the `migrate` command: create tables in a database based on the migration file

As we haven't touched `models.py`, we'll skip the first two steps in this topic. We'll cover those two steps in detail later.

Just for checking purposes, run the `makemigration` command. You'll see that nothing happens.

Command Line - INPUT

```
(d_env) project_d % | python manage.py  
makemigrations
```

Command Line - RESPONSE

```
No changes detected
```

However, there are several migration files already prepared for basic Django features. By running the `migrate` command, Django creates tables in a database using the preset migration files. The `migration` command is executed using the `manage.py` file like the shown below.

Command Line - INPUT

```
(d_env) project_d % | python manage.py  
migrate
```

When you run the command above, you'll see that migrations are applied to several migration files.

Command Line - RESPONSE

```
Operations to perform:  
  Apply all migrations: admin, auth, contenttypes, sessions  
Running migrations:  
  Applying contenttypes.0001_initial... OK  
  Applying auth.0001_initial... OK  
  Applying admin.0001_initial... OK  
  Applying admin.0002_logentry_remove_auto_add... OK  
  Applying admin.0003_logentry_add_action_flag_choices... OK  
  Applying contenttypes.0002_remove_content_type_name... OK  
  Applying auth.0002_alter_permission_name_max_length... OK  
  Applying auth.0003_alter_user_email_max_length... OK  
  Applying auth.0004_alter_user_username_opts... OK  
  Applying auth.0005_alter_user_last_login_null... OK  
  Applying auth.0006_require_contenttypes_0002... OK  
  Applying auth.0007_alter_validators_add_error_messages... OK
```

```
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying sessions.0001_initial... OK
```

When you execute the `runserver` command, you won't see the alert shown in the previous topic anymore.

Command Line - INPUT

```
(d_env) project_d % | python manage.py runserver
```

Command Line - RESPONSE

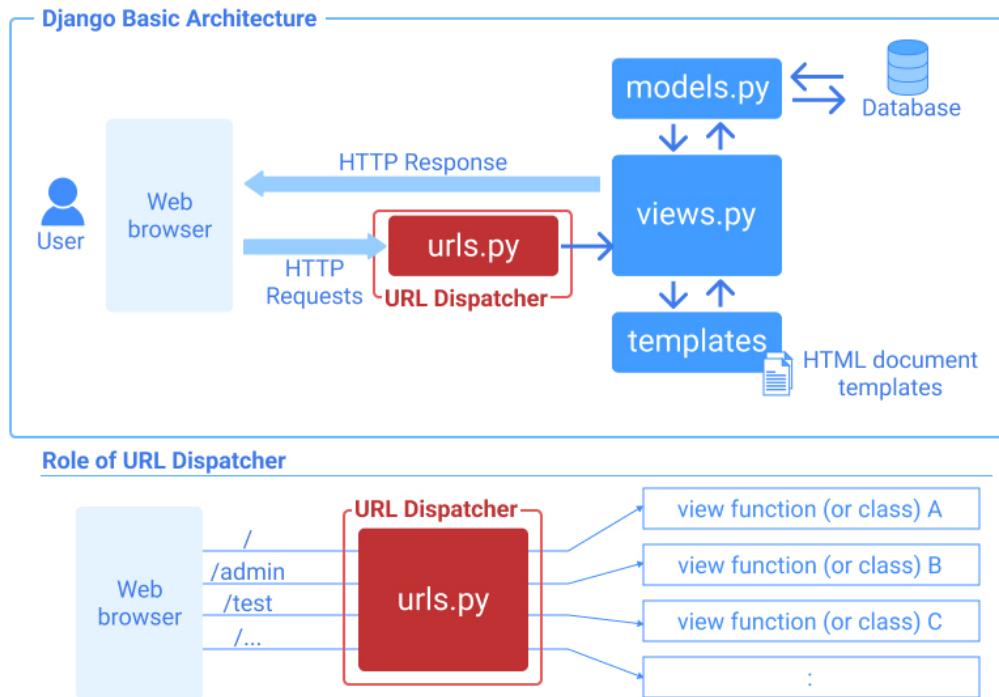
```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
March 19, 2023 - 02:40:30
Django version 4.1.7, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Note: Default Database for Django

The default database for Django is **SQLite**, which is open-source free database software. It is a lightweight, server-less database mainly used in small to medium-scale applications. It is better to use other databases, such as **MySQL** or **PostgreSQL**, for a large-scale application. To change the default database, you need to modify the `settings.py` file. We'll explain how to modify it in a later part of this course.

URL dispatcher – urls.py



The **URL dispatcher** is one of the key concepts in the Django architecture. The URL dispatcher maps HTTP requests to the view functions (or classes) specified in the URL patterns that are written in the urls.py file. You may not understand what this means if you are new to the concept. We'll explain the concept step by step.

The urls.py file

Let's open the *urls.py* file that is created when you start a Django project by running the `startproject` command. You can see that some code is already written in the file.

`config/urls.py`

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

urlpatterns = [
```

```
    path('admin/', admin.site.urls),  
]
```

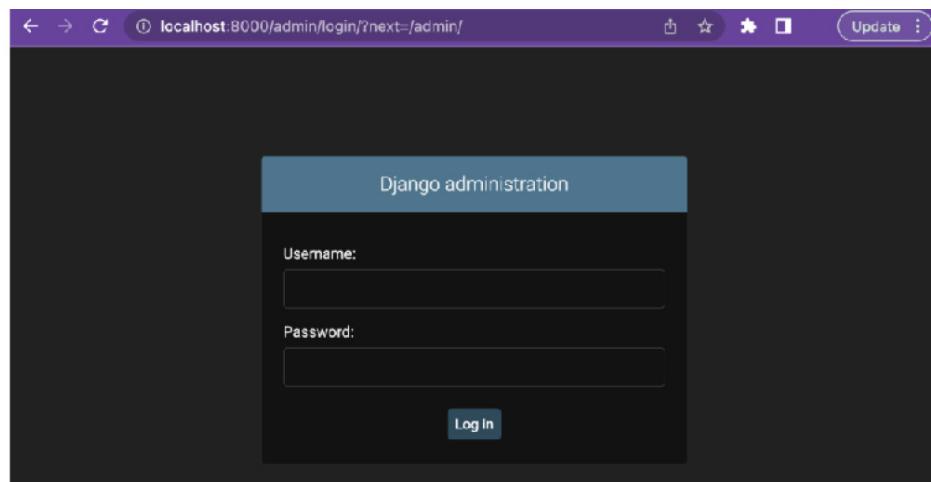
from ... import ...

This syntax is used when you want to use Python modules in the file. In this case, the `admin` module is imported from the `django.contrib` file, and the `path` module is imported from the `django.urls` file.

URL patterns

URL patterns are used for defining URLs for incoming HTTP requests and paths for the functions or classes to handle the requests. In this case, `'admin/'` is a URL for an HTTP request, and `'admin.site.urls'` is a function used to handle the HTTP request.

When the server is running, type `'localhost:8000/admin/'`. You can see the login view for the Django admin site as shown below.



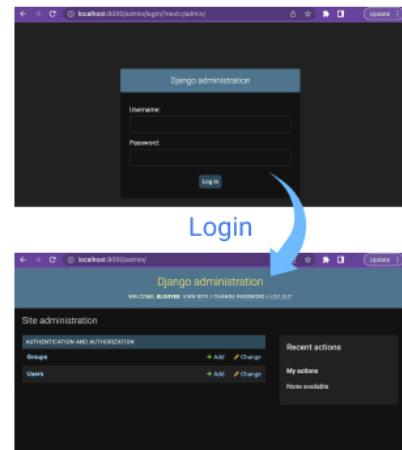
At this point, you cannot log in as no users have been created yet. We'll explain how to create a user to log in to the Django admin site later.

Create Superuser and Log In to Django Admin

```
python manage.py createsuperuser
```

Run the command from the project directory...

```
$ python manage.py createsuperuser
```



To log in to the Django admin site, you need to create a **superuser** by running the `createsuperuser` command.

The `createsuperuser` command

Run the command below to create a superuser for the Django admin.

Command Line – INPUT

```
(d_env) project_d % | python manage.py createsuperuser
```

You'll be asked to type a username, an email address, and a password. Django also checks password strength.

Command Line - INTERACTIVE

```
Username: ████████
Email address: ████████
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8
characters.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Log-in to Django admin page

You can log in to the Django admin page using the username and password you created.

Go to '`/localhost:8000/admin/`' in your browser, and type your username and password.

The screenshot shows the Django administration interface for the 'Users' model. The top navigation bar includes links for 'Home', 'Authentication and Authorization', and 'Users'. On the far right, there are links for 'WELCOME', 'BLOOVEE', 'VIEW SITE / CHANGE PASSWORD', and 'LOG OUT'. The main content area is titled 'Select user to change'. It features a search bar with a magnifying glass icon and a 'Search' button. Below the search bar, there is a table header with columns: Action, USERNAME, EMAIL ADDRESS, FIRST NAME, LAST NAME, and STAFF STATUS. A single row is visible in the table, showing the user 'bloohee' with the email 'bloohee@example.com'. To the right of the table, there is a 'FILTER' sidebar with three sections: 'By staff status' (with options All, Yes, No), 'By superuser status' (with options All, Yes, No), and 'By active' (with options All, Yes, No). The 'FILTER' sidebar has a dark blue header and light blue sub-section headers.

Create a normal user

You can also create a normal user on the admin site. On the **Users** page, you can find the user you created. You can create other users by clicking on the **ADD USER +** button on the top right.

Type a username and password on the page, and press the **SAVE** button to create a new user.

Add user

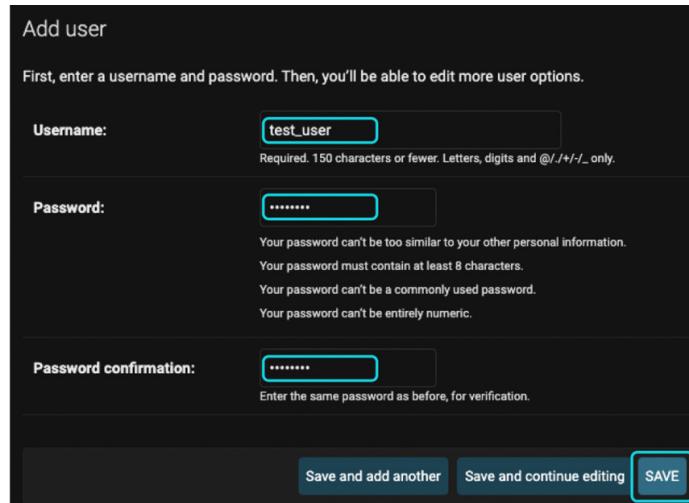
First, enter a username and password. Then, you'll be able to edit more user options.

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

Password: Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

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



After creating a user, you'll go to the **Change user** page. On the page, you can modify permission settings. As a default, the user is registered as a normal user who cannot access the Django admin site. If you want to make the user acces the admin site, check the **Staff** status. If you want to give the superuser status, which enables access to all resources of the app, check the **Superuser status**.

The user "test_user" was added successfully. You may edit it again below.

Change user

test_user

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

Password: algorithm: pbkdf2_sha256 iterations: 390000 salt: lu2Brz***** hash: Mb2CHj*****
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:

Permissions

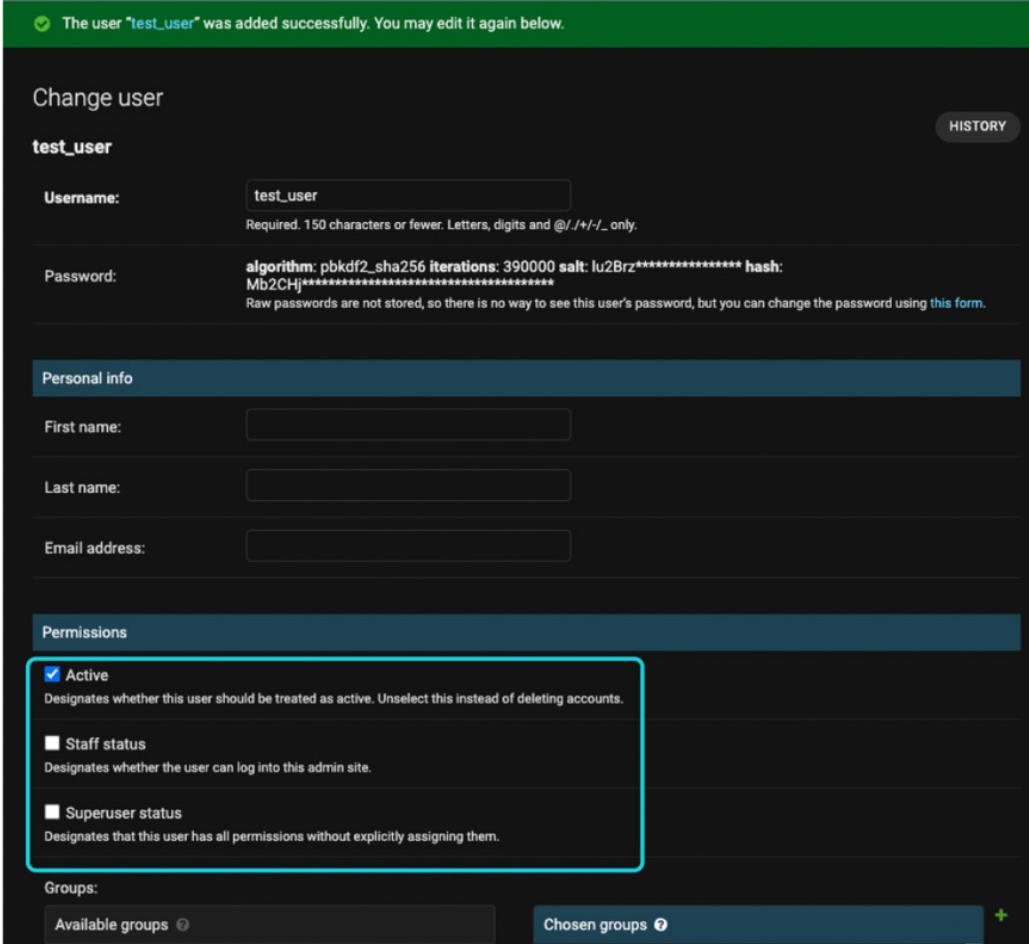
Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

Staff status
Designates whether the user can log into this admin site.

Superuser status
Designates that this user has all permissions without explicitly assigning them.

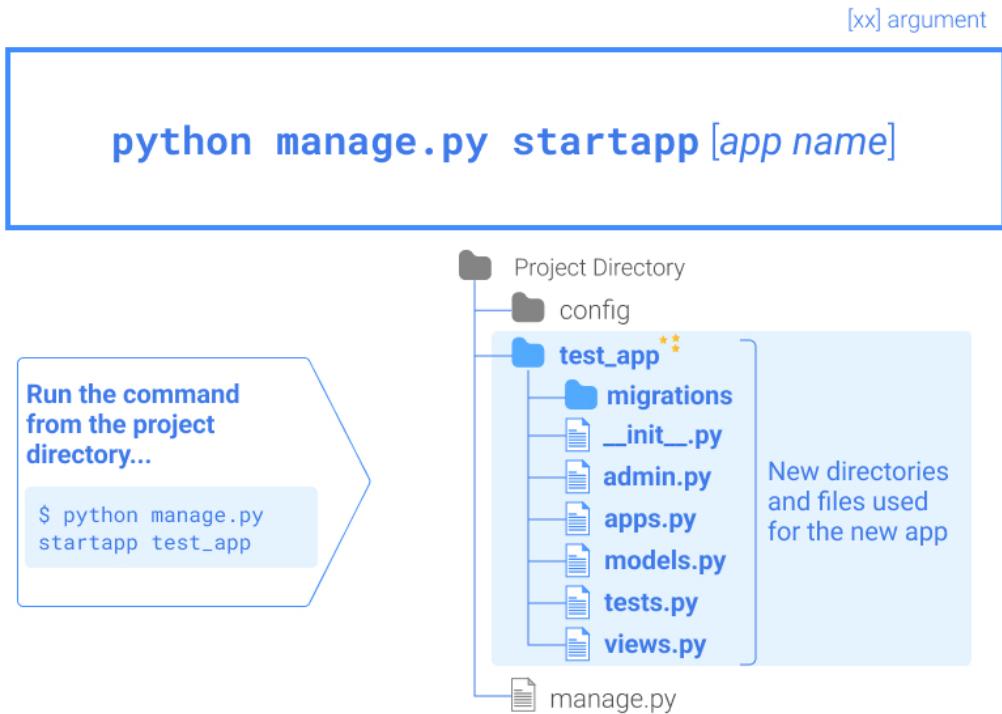
Groups:

Available groups [?](#) Chosen groups [?](#) [+](#)



If you go down on the **Change user** page, you'll see more detailed permission settings, including group settings. We don't explain these features here, but you can understand that Django has quite robust user permission features already. This is one of the key advantages of using the Django framework for your web app development.

Start App



The `django-admin startproject` creates only key files for the Django project configurations. To develop a Django application, you need to create key files used for actual application development. The `startapp` command is used to create necessary files for Django application development.

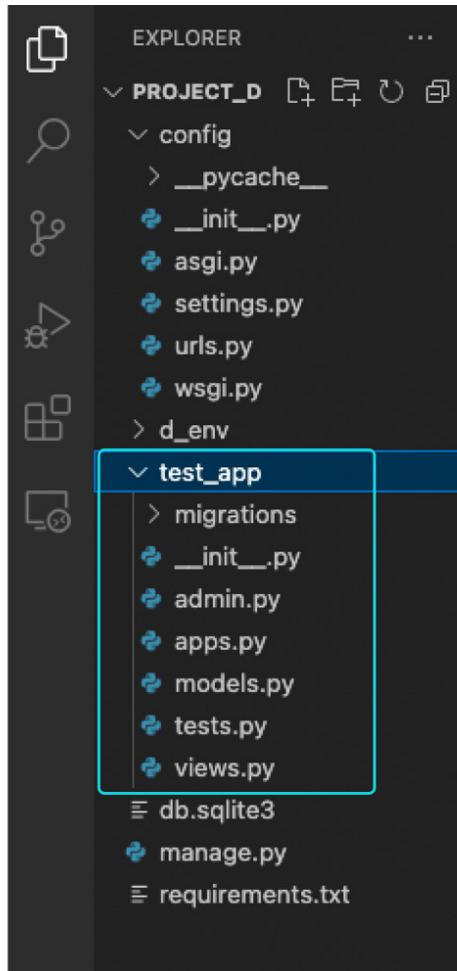
The startapp command

The `startapp` command is also executed through `manage.py`. When you run the command, you need to specify your application name. For example, run the command below to start an app named `test_app`.

Command Line - INPUT

```
(d_env) project_d % | python manage.py startapp test_app
```

When you run the command, you can see that many files are created under the new app directory.



migrations directory

When you run the `startapp` command, only the `__init__.py` file is created in this directory. This directory is used to store migration files. When you run the `makemigration` command, a new migration file is created.

`__init__.py`

As explained in the Start Django Project section, this is a particular file for Python programming used to mark a directory as a Python package. Often, that file has no content. In regular use, you don't need to touch this file.

admin.py (Important)

This file is used when you customize the Django admin site using the data created by the app.

apps.py

This file is used to customize the configuration of the app. If you are a beginner in Django, you don't need to worry about this file.

models.py (Very Important)

This file is used to design the database of the app. In the file, you can describe the structures of the database, including its data field and data type settings. We'll explain how to use this file later.

tests.py

This file is used for writing and running tests for your Django application.

views.py (Very Important)

This file is one of the most critical files in the Django app development. You'll be frequently touching this file. Functions or classes for handling **HTTP requests** are designed through this file.

Register the new app

By only running the `startapp` command, the Django project does not recognize the new app. To register the newly created app, you need to add the app name in the `settings.py` file.

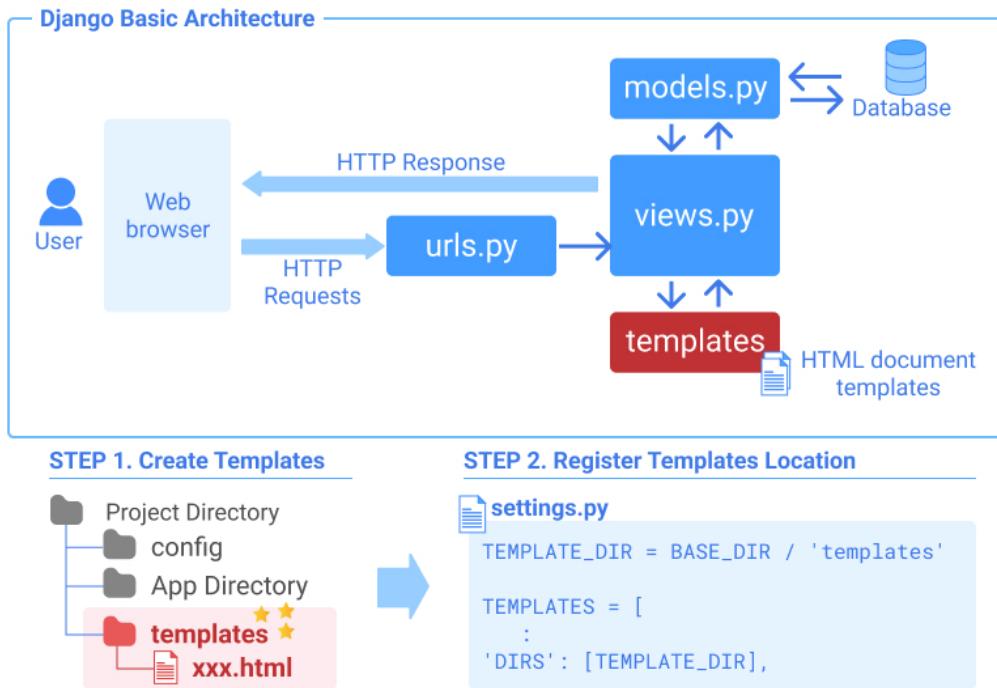
In our example, add '`test_app`' under the **INSTALLED_APPS** section of the `settings.py` file shown below. Do not forget to save the file to avoid making an error later on.

config/settings.py

```
:  
INSTALLED_APPS = [
```

```
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'test_app',
]
:
```

Create HTML Templates



When a Django app receives **HTTP requests**, the **URL dispatcher** dispatches them to functions or classes that are written in the `views.py` file to handle the requests. The `views.py` file focuses on logical responses. It doesn't handle visual representation for **HTTP responses**. To return the HTTP responses, you need HTML documents. In Django, the `templates` directory is used to save HTML documents linked to the `views.py` file. We'll explain how to make HTML templates for Django applications in this lesson. Details of `views.py` will be explained in the next section.

Create a 'templates' directory and an HTML document

As a default, the `templates` directory is not created yet. You need to create it to save HTML documents for the app. The best practice of the location of the `templates` directory is directly under

the project directory. If you are using Mac, run the commands below from the project directory.

Command Line - INPUT

```
(d_env) project_d % | mkdir templates  
(d_env) project_d % | cd templates
```

Command Line - INPUT

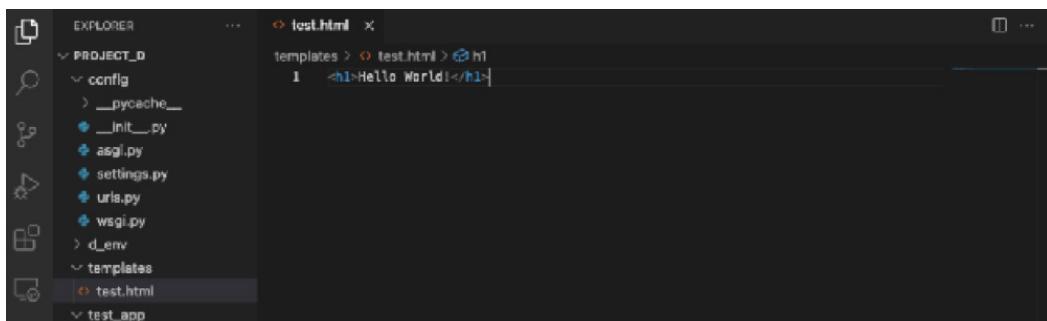
```
(d_env) project_d % | touch test.html
```

Or, you can create the directory and file from the left sidebar of VS Code.

Edit an HTML template

Let's create a classic message, "*Hello World!*" for the simple test purpose.

1. Select the test.html file in the VS Code.



2. Edit the file.

Type the code below. As this template will be used to test the **class-based view**, describe it in the HTML file. We'll explain the class-based view in the next section.

templates/test.html

```
<h1>Hello World! Class-based View Test</h1>
```

Register the 'templates' directory location in the `settings.py`

After creating the `templates` directory, you need to register the location of the directory in the `settings.py` file.

In the front part of the `settings.py` file, you can see that the path of `BASE_DIR` is defined. `BASE_DIR` is pointing to the project directory you created at the beginning of this chapter.

`BASE_DIR` gives you flexibility to change the project directory path. Regardless of the project directory absolute path, you can use `BASE_DIR` as the project directory path so that the file path stays the same even if you change your project directory location.

When you create a new directory right under the project directory, you can use `BASE_DIR / 'sub-directory'` as the directory path. The `sub-directory` part should be replaced with the actual directory name.

To add the `templates` directory you created under the project directory, add the **yellow** code part in the `settings.py`. The `templates` directory location should be written in the '`DIRS`' line in the `TEMPLATES` section. In this case, first define the file path variable in the front section. Then, use the file path variable in the '`DIRS`' line.

Do not forget to save the `settings.py` file when you edit it.

config/settings.py

```
from pathlib import Path
# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent

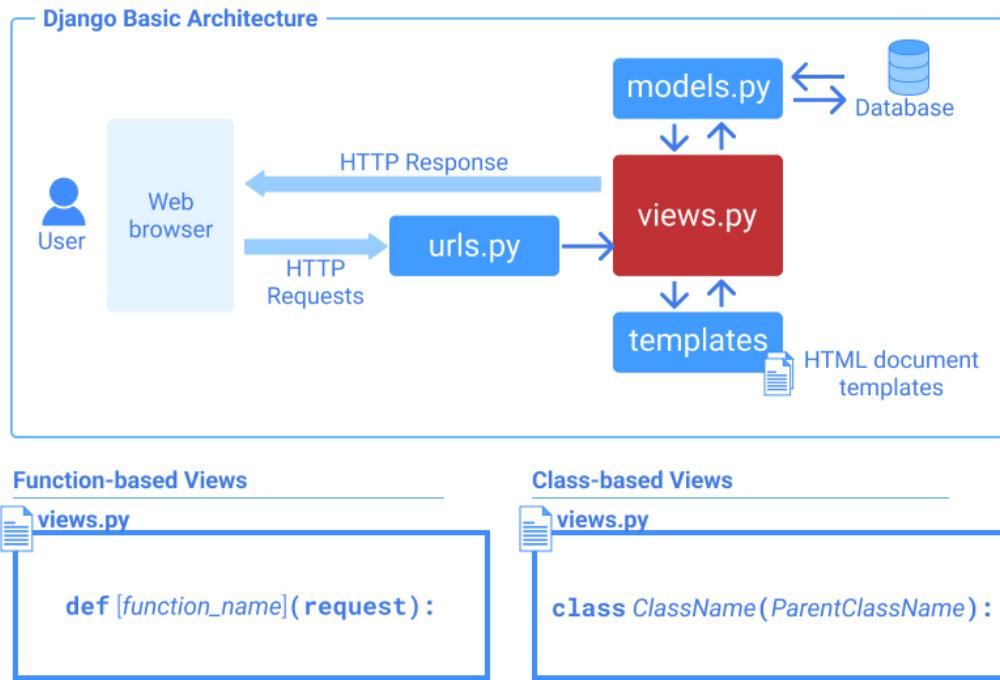
# template directory path
TEMPLATE_DIR = BASE_DIR / 'templates'

:

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR],
        'APP_DIRS': True,
```



Create Views



Views are the key part of the Django application that defines how to handle **HTTP requests**. When a Django app receives HTTP requests, the **URL dispatcher** handles the request by calling specified views for each URL. There are two types of views depending on how you write a view (logic): **1) Function-based views** and **2) Class-based views**.

Function-based Views

Function-based views are views written with Python functions instead of classes.

The way to write a function-based view follows Python function syntax.

- Function names are usually written in **snake_case** (no uppercase letters. To connect words, use underscores).

- Use **request** as an argument of the function

```
def [function_name](request):  
    logic to handle HTTP requests
```

The code below is an example of how to write a function-based view. The **yellow** parts of the code are added in this example.

In this case, we are using the HTTP response module. To import the module, add the second line.

- use '*function_view_test*' as the function name.
- describe "*Hello world! Function-based View Test*" with the <**h1**> tag as an HTML response.

`test_app/views.py`

```
from django.shortcuts import render  
from django.http import HttpResponseRedirect  
  
def function_view_test(request):  
    return HttpResponseRedirect('<h1>Hello world! Function-based View  
Test</h1>')
```

Class-based Views

Django provides predesigned **class-based views**. You can customize the views through inheritance. Class-based views have been more widely used recently as they enable saving coding time.

The way to write a class-based view follows Python class syntax.

- Class name is usually written in **PascalCase** (Capitalize all words).
- Use the **parent class name** as an argument of the class to inherit the parent class
- Add a **template file name** saved under the templates directory

- If you use a model in the view, add a **model name** defined under the model.py

```
class ClassName(ParentClassName):  
    template_name = 'HTML template file name'  
    model = ModelName  
    other logic if required
```

The code below is an example of how to write a class-based view. The **yellow** parts of the code are added in this example.

In this case, we are using *TemplateView*, which is one of the most simple predefined views. To import the view, add the third line.

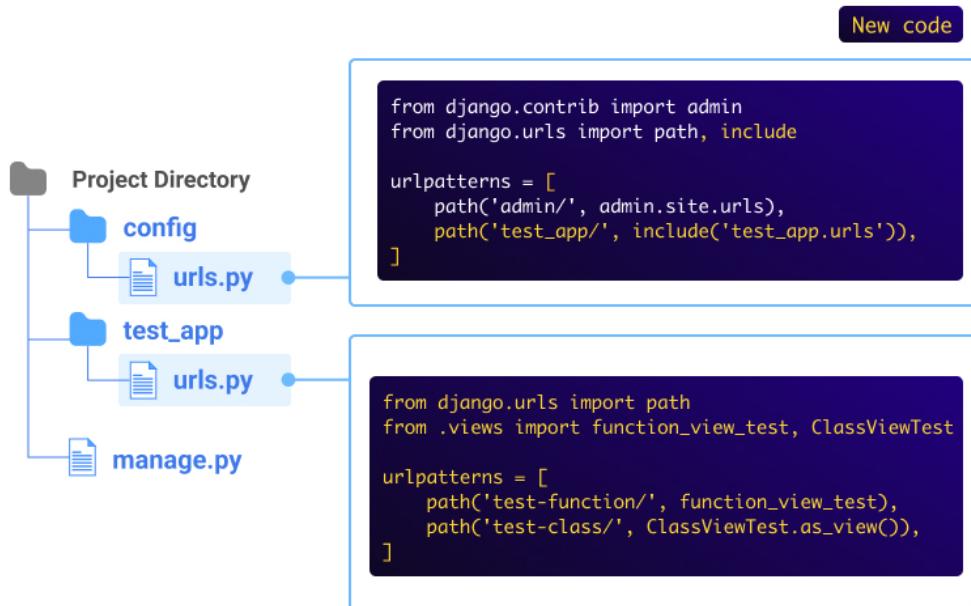
To make clear that this example code is used for class-based view tests, use "ClassViewTest" as the class name.

test_app/views.py

```
from django.shortcuts import render  
from django.http import HttpResponse  
from django.views.generic import TemplateView  
  
def function_view_test(request):  
    return HttpResponse('<h1>Hello world! Function-based View  
Test</h1>')  
  
class ClassViewTest(TemplateView):  
    template_name = 'test.html'
```

For more details about the class-based view, refer to Chapter 4.

Add URL Patterns



In the previous section, we prepared two views using a **function-based view** and a **class-based view**; however, those views are not linked with URLs yet. In this section, we'll add URL patterns for the views.

Add 'include' in the urls.py file

At this stage, there is only one *urls.py* created by the `django-admin startproject` command. The best practice to write URL patterns that are specific to the app is by creating a new *urls.py* file under the app directory. This is because you may add other apps to the project later. (We'll explain the app structure in the next section.)

The code below is an example of editing the main *urls.py* file. The **yellow** parts of the code are added in this example.

The key points here are:

1. import the `include` module
2. add new `urlpatterns` to include URLs written in the `test_app`

`config/urls.py`

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('test-app/', include('test_app.urls')),
]
```

Create a new `urls.py` file under the app directory

By the changes above, the path in the main `urls.py` is directed to the `urls.py` file in the `app` directory, which we haven't created yet.

To create the file, change the current working directory to the `app` directory. And create a new `urls.py` file.

Command Line - INPUT

```
(d_env) project_d % | cd ../test_app
```

Command Line - INPUT

```
(d_env) project_d % | touch urls.py
```

Or, you can create the directory and file from the left sidebar of VS Code.

The code below is an example of how to edit the `urls.py` file under the `app` directory.

The key points here are:

1. Import the `path` function that is used for defining URL patterns

2. Import the following views created in the previous section. Use a path '.views' to define the location of the views

- Import the `path` function that is used for defining URL patterns
- Import the following views created in the previous section. Use a path '`.views`' to define the location of the views
 - `function_view_test`
 - `ClassViewTest`

Add two URL patterns

- '`test-function/`' to call `function_view_test`
- '`test-class/`' to call `ClassViewTest` (you need to add `.as_view()` when you call a class-based view)

test_app/urls.py

```
from django.urls import path
from .views import function_view_test, ClassViewTest
urlpatterns = [
    path('test-function/', function_view_test),
    path('test-class/', ClassViewTest.as_view()),
]
```

Check the URLs

To check if the views are working, execute the `runserver` command. Make sure your are running the command from the project directory.

Command Line - INPUT

```
(d_env) project_d % | python manage.py runserver
```

When the server is running, go to the following URLs. You'll see the results in the images below.

- *localhost:8000/test-app/test-function/*
- *localhost:8000/test-app/test-class/*

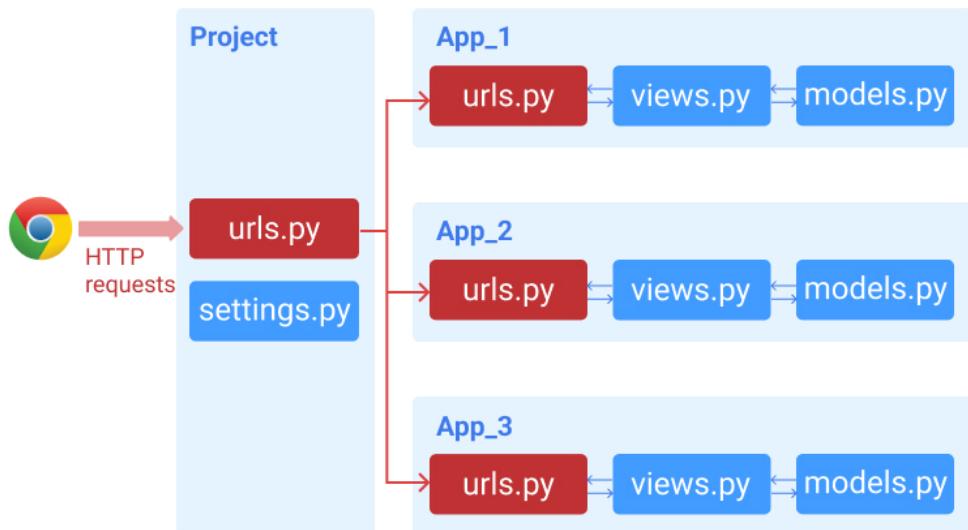
Function-based View



Class-based View



Project vs. App

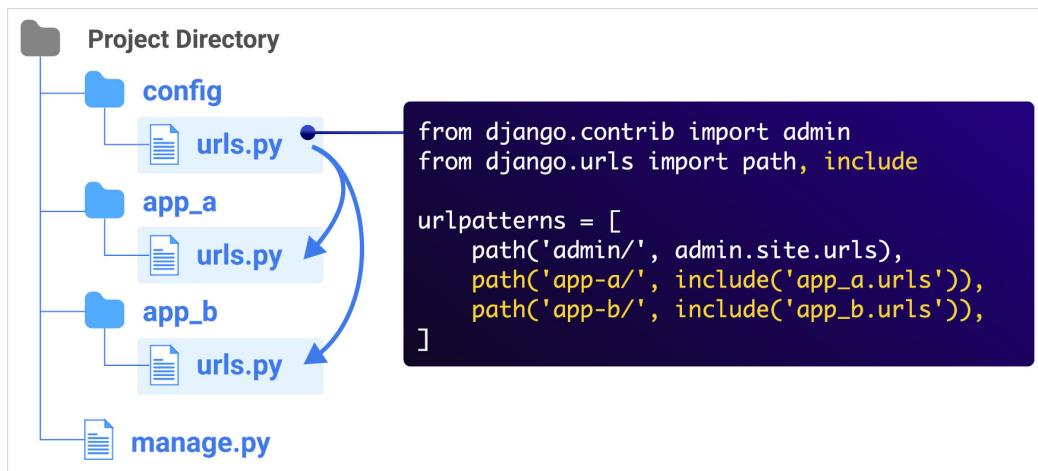


As explained in the previous sections, the `django-admin startproject` command mostly creates configuration files required to manage the project, while the `startapp` command creates key files to design Django applications. It is essential for you to understand that one Django project can have multiple apps. As you saw in the earlier section, several preset apps are already written. You can also add several customized apps of your own.

A **URL dispatcher** plays an important role in managing multiple apps under one Django project. The `urls.py` file created when you run the `django-admin startproject` command will be an entry point for HTTP requests. As explained in the previous section, the main `urls.py` file can

include other URLs written in other *urls.py* files created under the *app* directory.

The approach to connecting with more than one application is the same. You can add another line of URL patterns in the main *urls.py* file, like shown in the illustration below.



Chapter 03

Django Models and Databases

This chapter covers how to create a **database** for Django applications. In the first few sections, we'll explain the critical steps to create a database, and in the later part of this chapter, we'll explain how to edit the `models.py` file in detail.

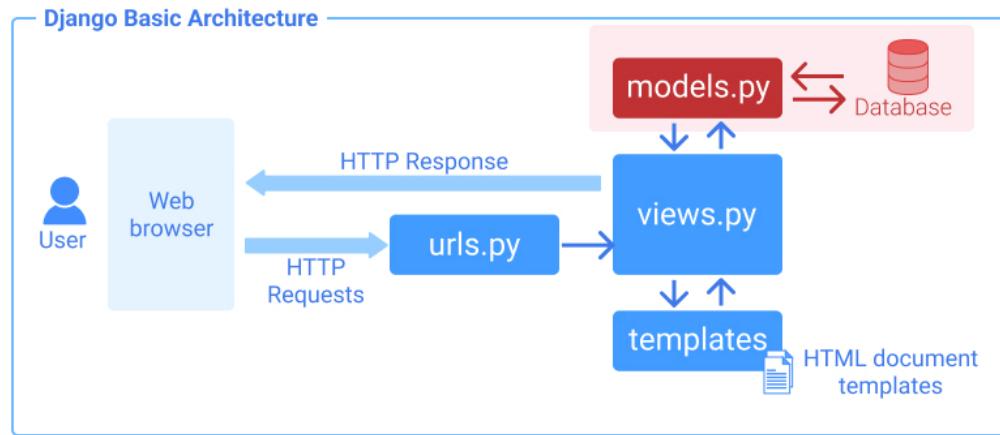
The following topics are covered in this chapter.

Topic

1. Create a Database in Django
2. Relational Database
3. Create Django Models
4. Makemigrations and Migrate
5. Add Models in Django Admin – `admin.py`
6. Change Display Name of Record Objects
7. Django Models – Data Field Type
8. Django Models – Field Options
9. Django Models – Help Text Option
10. Django Models – Choices Option

- [**11. Django Models – DateField with datetime Module**](#)
- [**12. Django Models – Relationship Fields**](#)
- [**13. Django Models – ID**](#)
- [**14. Django Models – ForeignKey \(OneToMany Relationship\)**](#)
- [**15. Django Models – OneToOneField**](#)
- [**16. Django Models – ManyToManyField**](#)

Create a Database in Django



3 STEPs to create a Django databasee

Create Django Models

- Design Django Models by editing `models.py`

Makemigration

- Make a migration file by running the `makemigration` command

Migrate

- Create database tables by running the `migrate` command

In the previous chapter, we briefly explained most parts of the **Django basic architecture** >— `urls.py` as a **URL dispatcher**, `views.py` for providing fundamental logic to handle **HTTP requests**, and `templates` for converting `views.py`'s output into `HTML` format.

`models.py` and **database** are the last critical parts of the Django basic architecture. To create a database in Django, `models.py` plays a critical role in providing database design information.

Key steps to create a database in Django

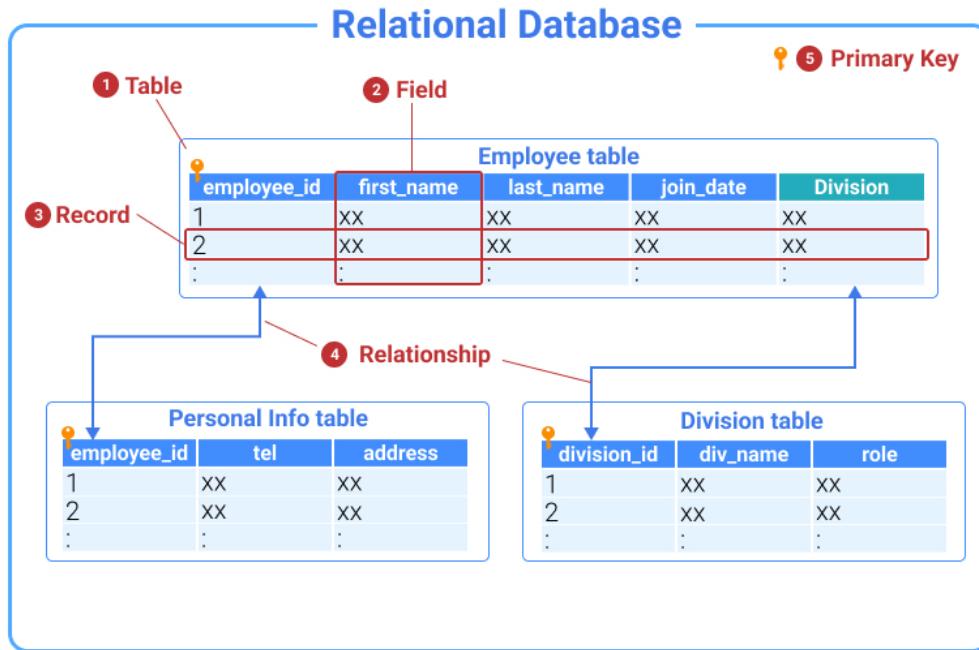
In Django, there are three steps to create a database.

1. Create **Django Models** by updating the `models.py` file
2. Create a migration file by running the `makemigration` command
3. Create database tables by running the `migrate` command

In addition to these three steps, you may want to modify the `admin.py` file to reflect a newly created database on the Django admin page.

In the following sections, we'll explain the key steps in detail.

Relational Database



Before jumping into **Django Models**, it is beneficial to review the key points of a **relational database**.

A database that Django officially supports is a relational database, which is the most commonly used database type. A relational database organizes data into columns and rows.

Key terms in Relational Database

There are five key terms in the relational database you need to understand.

1. Table

Tables are sets of data organized in rows and columns. A relational database consists of multiple tables. In the main figure example, a database consists of three tables – *Employee* table, *Personal Info* table, and *Division* table.

2. Field

Fields are a crucial part of a relational database storing data. Each field determines a data type that can store a specific type of data. We'll explain the data field types used in Django later.

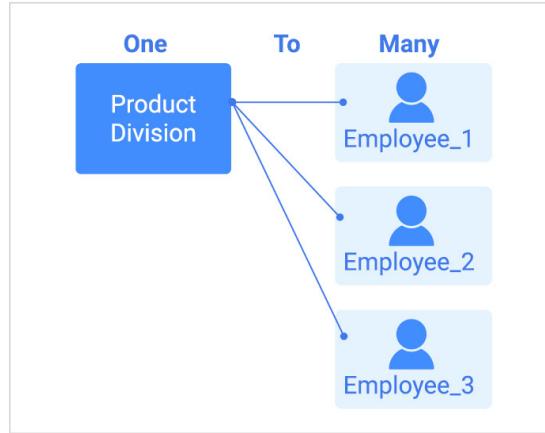
3. Record

A record is a set of values that are assigned to each field or column in a data table. In the example of the main figure, a record consists of data values that cover each field in the *Employee* table.

4. Relationship

A relationship defines a connection between two tables. There are three types of relationships.

- **One-to-Many Relationship:** This is the most frequently used relationship in the relational database. In this relationship, each record in one table can correspond to many records in another table, but not the other way around. For example, each division of a company consists of several employees, and each employee belongs to only one division.

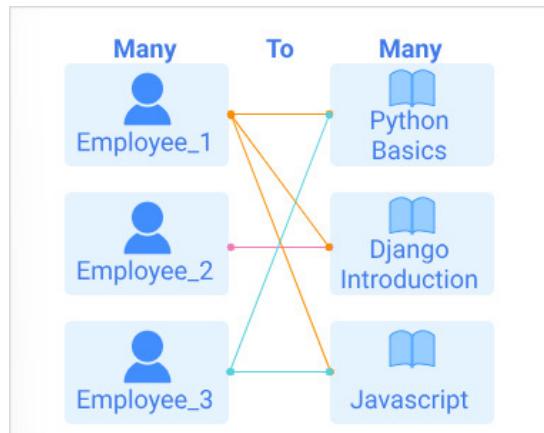


- **One-to-One Relationship:** In this relationship, each record in one table corresponds to one record in another table. In the main figure example, the relationship between the *Employee* table and the *Personal Info* table is one-to-one relationship. One-to-one relationship tables can be combined into one table, but in practice, there are reasons for separating tables from the data management point of view.



- **Many-to-Many Relationship:** In this relationship, each record in one table can correspond to many records in another table, and vice versa. For example, an employee can be enrolled in multiple digital learning courses, and each digital learning course can have multiple employees enrolled. To create a many-to-many relationship between

two tables, another table called **an intermediary join table (associative entity)** is created to join the two tables.



5. Primary Key

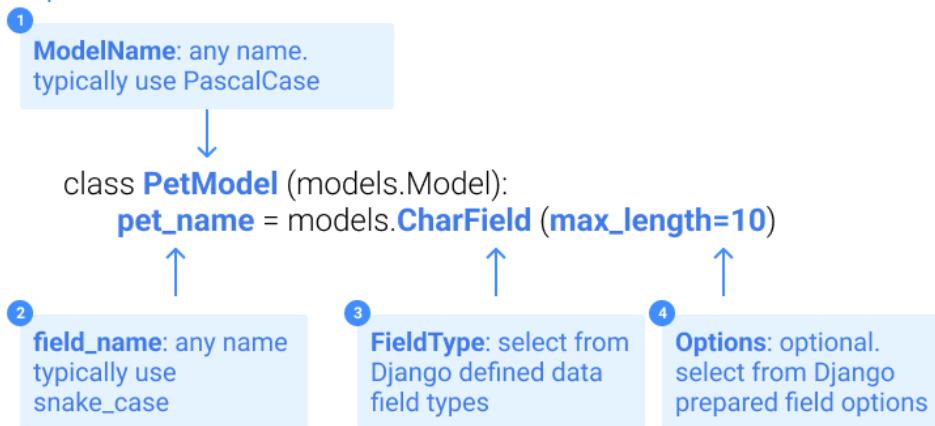
Primary key is defined as a field or column of a table that uniquely identifies each record in the table. The main role of the primary key is to define relationships with other tables in a relational database.

Create Django Models



```
class ModelName (models.Model):  
    filed_name = models.FieldType(option)
```

Example:



The first step of creating a database is designing a database structure. In Django, the database design is written in the *models.py* file. On this lesson, we'll explain how to edit *models.py*.

Edit *models.py*

Django models are defined as Python classes. When you run the `startapp` command, *models.py* is created.

By default, '`from django.db import models`' is already written at the top in *models.py*. This line of code imports a module called `models`, which will be the basis of the custom models you are designing.

To make a new model, you need to add a sentence to create a new class. The basic syntax for creating a class for a new model is shown in the main figure.



```
models.py

from django.db import models

class ModelName (models.Model):
    field_name_1 = models.FieldType(option)
    field_name_2 = models.FieldType(option)
    :
```

Add this part

There are four parts that you need to customize.

1. Model name

You can use any name (except for reserved words), but it should clearly represent the model that you are designing. One model becomes one data table in the database. You need to write in **PascalCase** because a Django model is defined as a Python class.

2. Field name

You can also use any name (except for reserved words) representing each model field (data table). Use **snake_case** for this part.

3. Field type

A field type defines the data type that each field can store. Django provides a list of field types, and you need to select one from the list. For example, **CharField** (Character Field) can store small- to large-sized strings. The list of field types will be explained later. A field type is written in **PascalCase**.

4. Field option

There are options you can add in each field. The field options give additional rules for the field. For example, you can allow the field to be `blank` using the blank option. Unlike the field type, the field option is written in **snake_case**.

Practice

Let's create a new app for this chapter and edit the `models.py` file. This chapter will guide you to create a database for employee learning programs.

1. Situation

HR skill development team is making a database to manage employees' digital skill learning courses. The team wants to manage the following information:

- which employees should sign which learning courses
- which employees should be prioritized for digital skill development

2. Create a new app "employee_learning"

Run the command below to create a new app under the same project created in the previous chapter.

Command Line - INPUT

```
(d_ev) peoject_d % | python manage.py startapp  
employee_learning
```

You can confirm that a new directory named `employee_learning` has been created under the project directory.

3. Register the new app in settings.py

When you create a new app, you always need to add it in the `settings.py` file so that Django can recognize the newly created directory as a new app.

Edit the `settings.py` file by adding the **yellow** line, as shown below.

`config/settings.py`

```
:  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'test_app',  
    'employee_learning',  
]  
:
```

4. Edit the models.py file

Edit the `models.py` file under the `employee_learning` app directory. As the first step, create an `Employee model` with one data field.

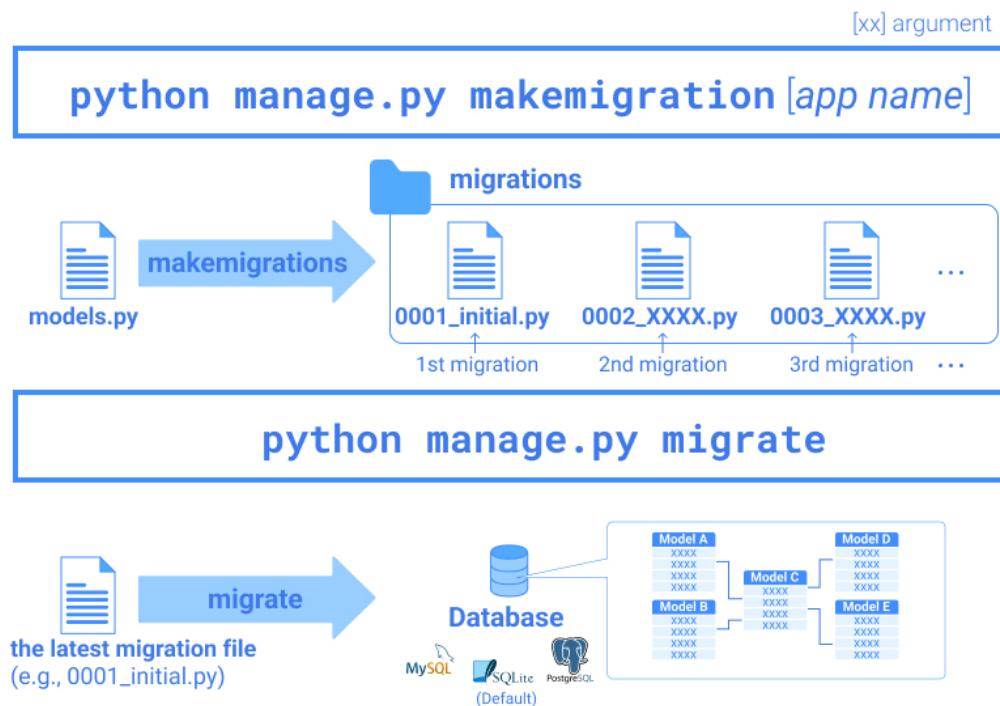
The four points below are the parts you need to customize in the basic syntax.

- **Model name:** `Employee`
- **Field name:** `name`
- **Field type:** `CharField`
- **Field option:** `max_length`

`employee_learning/models.py`

```
from django.db import models  
  
class Employee(models.Model):  
    name=models.CharField(max_length=25)
```

Makemigrations and Migrate



Once you have added a new model in the `models.py` file, you can start to create a database with two commands – `makemigrations` and `migrate`.

The makemigrations command

The `makemigrations` command is used for creating a migration file, which is a design file used for creating a database. For example, if you want to create a migration file for the `employee_learning` app created in the previous section, run the command below.

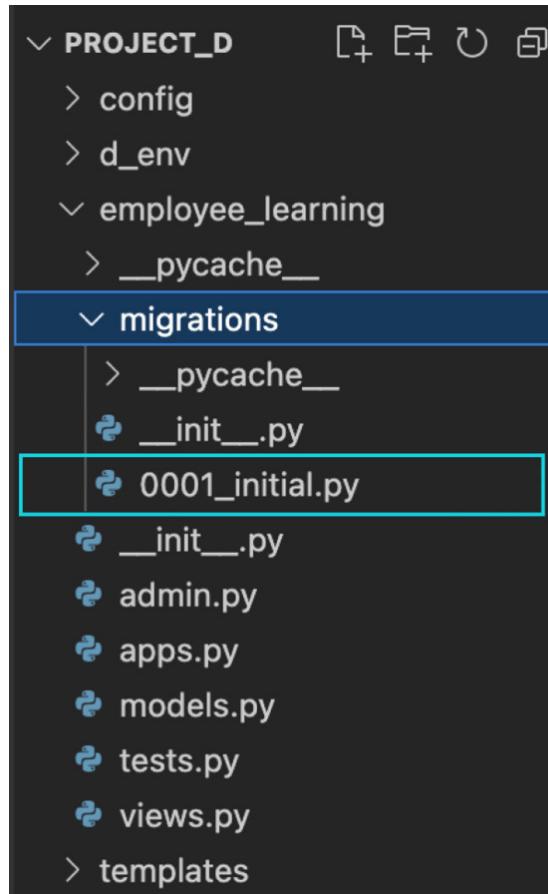
Command Line - INPUT

```
(d_env) project_d % | python manage.py makemigrations  
employee_learning
```

You can see that the migration file `0001_initial.py` has been created under the `migrations` directory.

Command Line - RESPONSE

```
Migrations for 'employee_learning':  
  employee_learning/migrations/0001_initial.py  
    - Create model Employee
```



The migration files are created when you update the *models.py* file and run the `makemigrations` command. The second time you run the `makemigrations` command, you'll see that a new file named *0002_xxx.py* is created. The *xxx* part of the filename is a simple description of key changes in the *models.py* file. Django captures the key changes and uses the information in the file name.

For example, add the *Division* model in the *models.py* file by adding the **yellow** line below and run the `makemigrations` command.

```
employee_learning/models.py  
from django.db import models
```

```
class Employee(models.Model):
    name=models.CharField(max_length=25)

class Division(models.Model):
    div_name=models.CharField(max_length=25)
```

Command Line - INPUT

```
(d_env) project_d % | python manage.py makemigrations
employee_learning
```

You can see that the migration file *0002_division.py* is created under the migrations directory.

Command Line - RESPONSE

```
Migrations for 'employee_learning':
  employee_learning/migrations/0002_division.py
    - Create model Division
```

The benefit of having two commands (`makemigrations` and `migrate`) in the database migration is that you can carefully manage the process. The `makemigrations` step is especially helpful when you are updating the model after the database was populated with data.

For example, when new data fields are added to the *models.py* file, the command initiates the interactive mode and asks how to handle the data for new fields.

Note: App name for the `makemigrations` command

You can run the `makemigrations` command without specifying an app name to create a migration file, but it is recommended to specify an app name to avoid potential future troubles.

For example, when you are making two apps in the same project, you may have two *models.py* files in each of the two apps: one with a completed model design and the other still in process of being designed. In that case, if you run the `makemigrations` command without specifying an app name, Django will create migrations files for both apps. A migration file created by the

incomplete `models.py` file may cause a problem in the database, which can be irreversible.

The migrate command

As explained in the previous chapter, the `migrate` command creates or updates a database using a migration file prepared by the `makemigrations` command.

As we have already updated migration files, run the `migrate` command again.

Command Line - INPUT

```
(d_env) project_d % | python manage.py migrate
```

You can see that two migration files were migrated into the database.

Command Line - RESPONSE

```
Operations to perform:
```

```
  Apply all migrations: admin, auth, contenttypes,  
  employee_learning, sessions, test_app
```

```
Running migrations:
```

```
  Applying employee_learning.0001_initial... OK  
  Applying employee_learning.0002_division... OK
```

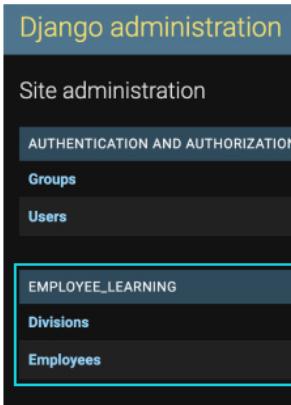
Add Models in Django Admin – admin.py

admin.py

```
from .models import ModelName  
admin.site.register(ModelName)
```

Add
this
part

```
admin.py  
from django.contrib import admin  
  
# Register your models here.  
from .models import Division  
from .models import Employee  
  
admin.site.register(Division)  
admin.site.register(Employee)
```



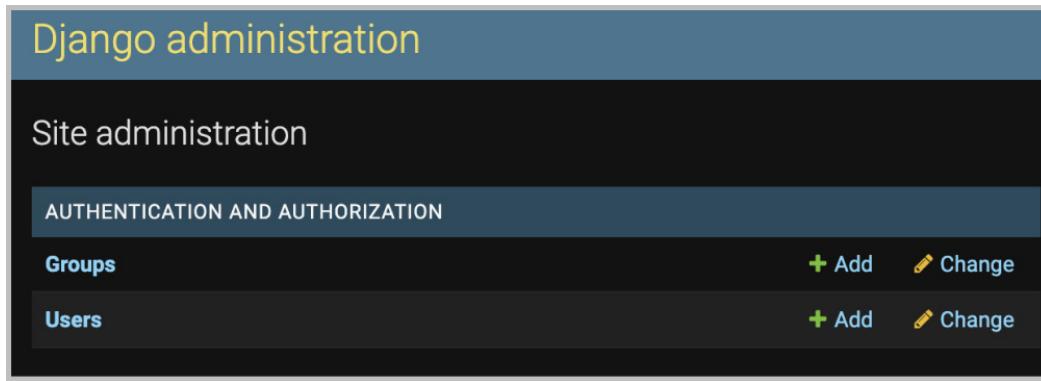
Even though you have migrated Django models into a database, you cannot see it instantly. The quickest way to see the database is through the **Django admin** site. To see your database in the Django admin site, you need to add some code in the *admin.py* file under the app directory.

To check the status, run the `runserver` command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py runserver
```

Go to `localhost:8000/admin/` in your browser. You can confirm that there is no model besides the models already prepared (i.e., Groups and Users).



Edit admin.py

Adding new models to the `admin.py` file is straightforward.

You need to import a model you want to add and register the model. You can register the two models we created in the previous sections by adding the `yellow` line below.

`employee_learning/admin.py`

```
from django.contrib import admin

# Register your models here.
from .models import Employee
from .models import Division

admin.site.register(Employee)
admin.site.register(Division)
```

Go to `localhost:8000/admin/` again. You'll see that the two models are added to the admin site. You can manage the process to update the `admin.py` file without stopping and re-running the `runserver` command.

Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION

[Groups](#)

[+ Add](#) [Change](#)

[Users](#)

[+ Add](#) [Change](#)

EMPLOYEE_LEARNING

[Divisions](#)

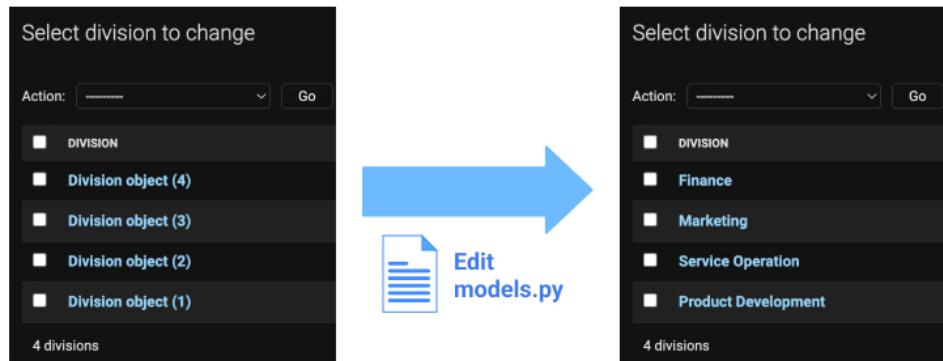
[+ Add](#) [Change](#)

[Employees](#)

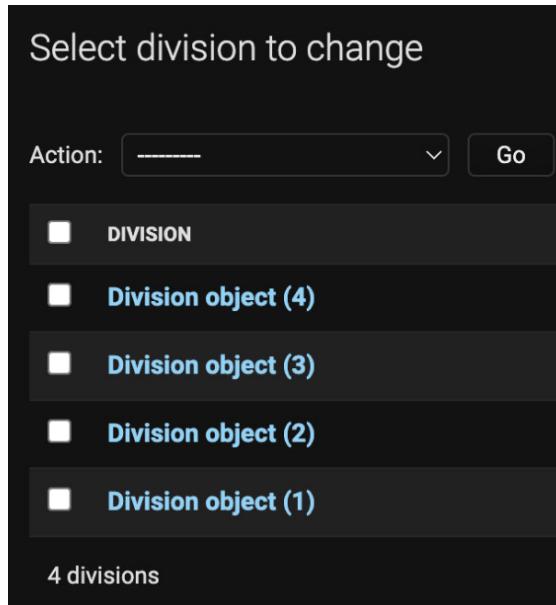
[+ Add](#) [Change](#)

Change Display Name of Record Objects

```
models.py
def __str__(self):
    return self.filed_name
```



The records displayed in the **Django admin** site are usually **Model Name + object (#)**, which is not user-friendly. For example, the data records of the *Division* model that we created on the previous page are shown below.



In Django, you can customize the record display name of each model by adding a simple code in *models.py*.

The `__str__(self)` method

The simplest way to change the record display name is by using one of the field names. For example, in the *Division* model, we can use field value in the `div_name` field.

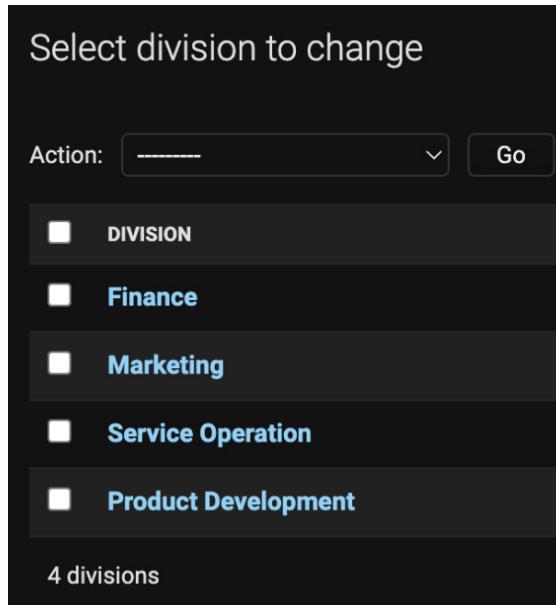
Edit the *models.py* file by adding the **yellow** line, as shown below.

`employee_learning/models.py`

```
class Division(models.Model):
    div_name=models.CharField(max_length=25)

    def __str__(self):
        return self.div_name
```

Check the admin site after saving the *models.py* file. You can see that the record names have been changed.



Customize record names

By using **f-string formatting**, you can customize the record display name further. For example, if you want to show the ID and employee name as a record display name of the *Employee* model, edit *models.py* by adding the **yellow** line below.

`employee_learning/models.py`

```
class Employee(models.Model):
    name=models.CharField(max_length=25)

    def __str__(self):
        return f'{self.id} : {self.name}'
```

Even though you don't create the `id` field when you create a model, Django automatically creates the `id` field, which is used as the model's **primary key**. You can call `id` by '`self.id`'. `self` is the *Employee* model class itself, and `id` is an **attribute** of the class. '`self.name`' is the `name` attribute of the *Employee* model class itself. The `name` attribute is created when you design the model as a field name.

After saving the *models.py* file, go to the Django admin site. You can see that the records of the *Employee* model have been changed like shown below.

Select employee to change

Action:

	EMPLOYEE
<input type="checkbox"/>	4 : Floyd Miles
<input type="checkbox"/>	3 : Eleanor Pena
<input type="checkbox"/>	2 : Bessie Cooper
<input type="checkbox"/>	1 : Leslie Alexander
4 employees	

Django Models – Data Field Type

	Field	Stored Data
Strings	<code>CharField</code>	strings (need to specify max_length)
	<code>TextField</code>	a large text
	<code>SlugField</code>	strings of alphabets, numbers, underscores or hyphens
	<code>URLField</code>	a URL (validated by URLValidator)
	<code>EmailField</code>	an email address (validated by EmailValidator)
Logic	<code>BooleanField</code>	True / False
Numbers	<code>IntegerField</code>	an integer
	<code>PositiveIntegerField</code>	a positive integer
	<code>FloatField</code>	a floating-point number
	<code>DecimalField</code>	a fixed-precision decimal number
	<code>AutoField</code>	an integer that automatically increments
Date & Time	<code>DateField</code>	a date
	<code>DateTimeField</code>	a date and time
	<code>TimeField</code>	a time
Files	<code>ImageField</code>	image file information (uploaded)
	<code>FileField</code>	file information (uploaded)

Fields are a crucial part of a **relational database** storing data. Each field determines a data type that can store a specific type of data. Django provides several field types. A field type is written in **PascalCase**. Some field types require specific arguments. For example, `max_length` is required for `CharField`. In this section, we'll explain key data field types.

1. Field types for strings

CharField

This field type is one of the most frequently used field types. This field can store small-to-large strings.

Key Arguments

- `max_length`: Required argument. Without this argument, Django gives an error message. Using this argument, you can specify the maximum length of the field.

TextField

This field is used to store a large text.

Key Arguments

- `max_length`: Optional argument. Using this argument, you can specify the maximum length of the field.

SlugField

This field is used to store strings of alphabets, numbers, underscores, or hyphens, which are generally used in URLs.

Key Arguments

- `max_length`: Optional argument. If you don't specify `max_length`, a default of 50 is used.

URLField

This field is used to store a URL validated by a URL validator.

Key Arguments

- `max_length`: Optional argument. If you don't specify `max_length`, a default of 200 is used.

2. Field types for Logic

BooleanField

This field is used to store a `True` or `False` value. In a form input, this field usually becomes a check box input.

3. Field types for Numbers

IntegerField

This field is used to store an integer. According to the Django documentation, Values from `-2147483648` to `2147483647` are safe in all databases supported by Django.

PositiveIntegerField

This field is used to store a positive integer or zero value. According to the Django documentation, Values from `0` to `2147483647` are safe in all databases supported by Django.

FloatField

This field is used to store a floating-point number represented in Python by a float instance.

DecimalField

This field is used to store a fixed-precision decimal number, represented in Python by a decimal instance.

Key Arguments

- `decimal_places`: Required argument. Using this argument, you can specify the number of decimal places to store the number.
- `max_digits`: Required argument. Using this argument, you can specify the maximum number of digits allowed in the number. This number must be greater than or equal to `decimal_places`.

Note: Floating-point number vs. decimal number

Floating-point numbers in **FloatField** are handled in binary numbers. In some cases, calculations using this field may not be very accurate, but the data processing speed tends to be faster.

Decimal numbers in **DecimalField** are handled in decimal numbers. Calculations using this field are more accurate, but the data processing speed tends to be slower.

AutoField

This field is used to store an integer that automatically increments. This type of field is used for primary key IDs. Django gives each model an auto-incrementing primary key by default, so you usually won't need to use this field type.

4. Field types for Date & Time

DateField

This field is used to store a **date**, represented in Python by a `datetime.date` instance.

Key Arguments

- `auto_now_add`: Optional argument. This option is often used for the `created_at` column in a database table. When this argument is True, the value in the field is set with the current date when the record object is first created.
- `auto_now`: Optional argument. This option is often used for the `updated_at` column in a database table. When this argument is True, the value in the field is updated with the current date when the record object is updated and saved.

DateTimeField

This field is used to store **date and time**, represented in Python by a `datetime.datetime` instance.

Key Arguments

- The `auto_now_add` and `auto_now` options are optional arguments for this field type.

TimeField

This field is used to store **time**, represented in Python by a `datetime.time` instance.

Key Arguments

- The `auto_now_add` and `auto_now` options are optional arguments for this field type.

Tips: Editable timestamp

When `auto_now_add` or `auto_now` argument is True, you won't be able to modify the field (`DateField` or `DateTimeField`) as it is used for the auto-generated timestamp. If you want to modify the field, you need to try another implementation using imported modules like the ones below.

- **For DateField:** `default=date.today` - from `datetime.date.today()`
- **For DateTimeField:** `default=timezone.now` - from `django.utils.timezone.now()`

We'll explain this in the later section of this chapter.

5. Field types for Files

ImageField

This field is used to store image file information. The Pillow library is required to use this field type.

Key Arguments

- `upload_to`: Required argument. Using this argument, you can specify the location (subdirectory) to save an uploaded image file. This argument only specifies a subdirectory path. The main directory for uploading files is defined in the `settings.py` file – `MEDIA_ROOT` and `MEDIA_URL`. If you are not creating a subdirectory, you can use a blank for the option value (`upload_to=' '`). We'll explain how to edit `settings.py` for media file handling later.

- `height_field`: Optional argument. Using this argument, you can specify the height of the image.
- `width_field`: Optional argument. Using this argument, you can specify the width of the image.

Note: Pillow

Pillow is a **Python Imaging Library (PIL)** fork that adds image processing capabilities to the Python interpreter. When you use `ImageField` in Django, you need this library in your app.

FileField

This field is used to store file information. Using this field type, you can handle other-than-image files.

Key Arguments

- There are optional arguments for this field type, such as `upload_to` and `storage`.

Django Models – Field Options

	Field Options	Roles
Data Validation	<code>blank</code>	If True, the field is allowed to be blank
	<code>unique</code>	If True, this field must be unique throughout the table.
	<code>validators</code>	Using this option, you can add custom data validations
Data Input	<code>choices</code>	Using this option, you can alter the field input to choices from a defined list
	<code>default</code>	This option defines a default value when a new record object is created.
Data Display	<code>help_text</code>	This option defines help text shown beside the input form of a field
	<code>verbose_name</code>	Using this option, you can customize the field name displayed in browsers
Database Handling	<code>primary_key</code>	This option defines a primary key field. The default is set at the auto generated id field
	<code>db_column</code>	This option defines the database column name, which is the field name by default.

There are options you can add in each field. The field options give additional rules for the field. Different from the field type, the field option is written in **snake_case**.

1. Data validation related options

`blank`

The field is allowed to be blank if `blank=True`. If you don't set this option for the field, you cannot leave the field blank when you make an input for it.

`unique`

The field value must be unique in the table if `unique=True`. If you don't set this option for the field, you can put the same

value in different records.

`validators`

Using this field, you can add a custom validation. To make the custom validation, you need to create validation rules with a function or class. Use the name of the function or class as a value of the option (e.g., `validators=[function_name]`).

2. Data input-related options

`choices`

You can alter the field input to choices from a defined list using this option. Usually, the field with this option will be displayed in a drop-down list format in browsers when you deploy the app.

`default`

Using this option, you can set a default value for the field that is used when a new record object is created.

3. Data display-related options

`help_text`

Using this option, you can set help text that is shown beside the input form of the field when you deploy the app.

`verbose_name`

Using this option, you can customize the field name displayed in browsers when you deploy the app. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.

4. Data handling related options

`primary_key`

When you create a model, Django automatically adds an `id` field to each model, which is used as the primary key for the model. Using this option, you can change the primary key to the field with this option.

`db_column`

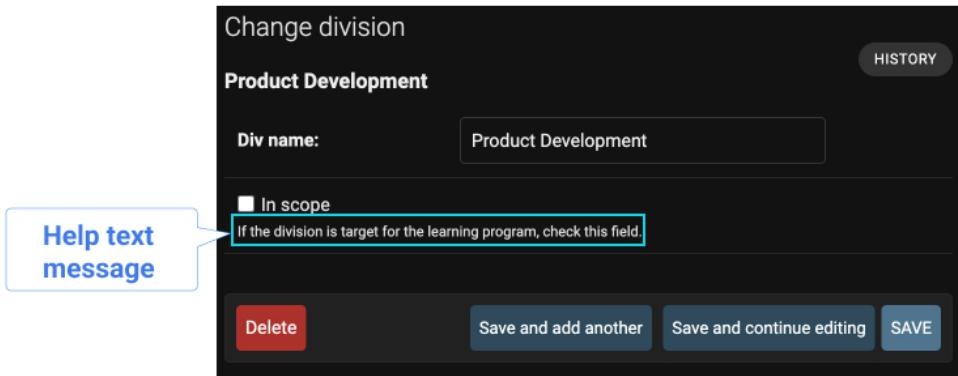
Django uses a field name for the column name in the database table. Using this option, you can customize the column name.

Django Models – Help Text Option



```
models.py

filed_name = models.FieldType(help_text="text message")
```



Using the `help_text` field option, you can add a help text message in the Django admin site for the field. Its execution is straightforward. Add the `help_text` option with a text message you want to show.

Practice

Objective:

Add a new field in an existing model after data entry

As we already added some data in the models, when we add new fields in the existing model, we need to consider how to handle the missing data for the new fields in the existing model.

In this practice, we'll explain how to handle it while implementing the `help_text` option.

1. Edit models.py

Edit the `Division` model by adding a new field named `in_scope`. This field is expected to be used to select the target division of the digital skill learning program. Use the `BooleanField` with the `help_text` option. Edit the `models.py` file by adding the **yellow** line, as shown below.

`employee_learning/models.py`

```
class Division(models.Model):
    div_name=models.CharField (max_length=25)
    in_scope=models.BooleanField (help_text="If the
        division is target for the learning program,
        check this field.")

    def __str__(self):
        return self.div_name
```

2. Run the makemigrations command

Run the `makemigration` command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py
makemigrations employee_learning
```

You'll see an alert message saying, "...*impossible to add a non-nullable field 'in_scope' to division without specifying a default...*". This is because the data for the new field of the existing records are missing.

Command Line - INTERACTIVE

```
It is impossible to add a non-nullable field  
'in_scope' to division without specifying a  
default. This is because the database needs  
something to populate existing rows.
```

```
Please select a fix:
```

- 1) Provide a one-off default now (will be set on
all existing rows with a null value for this
column)
- 2) Quit and manually define a default value in
models.py.

```
Select an option:
```

Answer 1 at this time. Then, the prompt will ask what data should be filled in.

Command Line - INTERACTIVE

```
Select an option: 1
```

```
Please enter the default value as valid Python.
```

```
The datetime and django.utils.timezone modules are  
available, so it is possible to provide e.g.  
timezone.now as a value.
```

```
Type 'exit' to exit this prompt
```

```
>>>
```

As the field type of the `in_scope` field is `Boolean`, type `False`. You'll see that a new migration file is created.

Command Line - INTERACTIVE

```
>>> False  
Migrations for 'employee_learning':  
    employee_learning/migrations/0003  
        _division_in_scope.py  
            - Add field in_scope to division
```

3. Run the `migrate` command

Finally, run the `migrate` command.

Command Line - INPUT

```
(d_ev) project_d % | python manage.py migrate
```

You'll see that the migration is successfully completed.

Command Line - RESPONSE

Operations to perform:

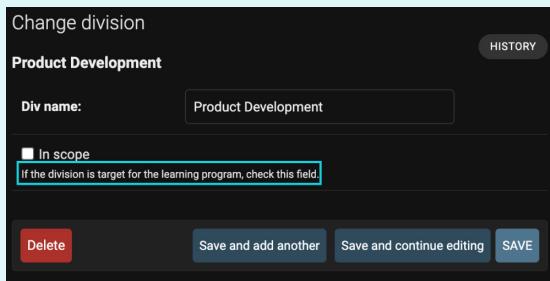
Apply all migrations: admin, auth, contenttypes, employee_learning, sessions, test_app

Running migrations:

Applying employee_learning.0003_division_in_scope... OK

4. Check the Django admin site

Go to localhost:8000/admin/. Open one of the records of the *Division* model. You can see that the *in_scope* field with the helping text message has been added to the Division model. Also, the checkbox is not checked, as we put "False" as the default value for existing records.



Django Models – Choices Option

```
models.py
Class ModelName(models.Model)
    LIST=[('H','High'), ('M','Medium'), ('L','Low'),]
    field_name = models.CharField(..., choices=LIST, ...)
```

List of tuples
class variable (any name)
LIST=[
 ('H','High'),
 ('M','Medium'),
 ('L','Low'),
]
human-readable name displayed in browsers

Change employee
Floyd Miles
Employee Name:
Floyd Miles
Learning Priorities:
High
✓ Medium
Low
Delete Save and add another Save and continue editing SAVE

Using the choices option, you can make a pull-down list input. There are two parts of coding to execute this.

1. Define list items with tuples

Usually, use a class variable to define list items. The list should be a list of **tuples**. The first element in each tuple is the actual value to be set in the model. The second element is a human-readable name to be displayed in browsers.

2. Add the choices option

We normally use `CharField` for the choices option. Use the defined list variable as the value of the option.

Practice

Objective:

Learn choices, verbose_name, default options

In this practice, we'll explain how to execute the `choices` option along with the `default` and `verbose_name` options.

1. The choices option

Edit the `Employee` model by adding a new field named priority. This field is expected to be used to select priority employees for the digital skill learning program.

To implement the `choices` option, edit the `models.py` file by adding the **yellow** line, as shown below.

`employee_learning/models.py`

```
class Employee(models.Model):
    PRIORITIES=[('H', 'High'), ('M', 'Medium'),
    ('L', 'Low'),]
    name=models.CharField(max_length=25, verbose_name
    ="Employee Name")
    priority=models.CharField(max_length=1,
    choices=PRIORITIES)
    def __str__(self):
        return self.name
```

2. The default option

You can specify the default value for the field using the `default` option. In this case, we use "M" as the default value.

`employee_learning/models.py`

:

```
priority=models.CharField(max_length=1,  
choices=PRIORITIES, default="M")  
:
```

3. The `verbose_name` option

The `verbose_name` option is used to adjust how to display the field name in browsers. If you don't specify it, the displayed field name is *Priority*. Using a simple word is suitable for computers, but it may not be clear enough for humans. In this case, we'll change the display name to "*Learning Priorities*" by adding the `(yellow)` line below.

`employee_learning/models.py`

```
:  
priority=models.CharField(max_length=1,  
verbose_name="Learning Priorities",  
choices=PRIORITIES, default="M")  
:
```

4. Execute the changes in the database

Execute the `makemigrations` and `migrate` command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py  
makemigrations employee_learning  
(d_env) project_d % | python manage.py migrate
```

5. Check the admin site

Run the `runserver` command and go to the admin page. You can confirm that the three options (1. `choices`, 2. `default`, 3. `verbose_name`) are appropriately executed.

Change employee

Floyd Miles

Employee Name:

Floyd Miles

Learning Priorities:

High
✓ Medium
Low

Delete Save and add another Save and continue editing SAVE

HISTORY

Django Models – DateField with datetime Module

```
models.py

import datetime
:
field_name = models.DateField(default=datetime.date.today, ...)
```

The screenshot shows the Django Admin interface for a 'Change employee' entry. The employee name is 'Floyd Miles'. Under 'Learning Priorities', there is a dropdown menu set to 'Medium'. At the bottom, the 'Data Registration Date' is displayed as '2023-03-23' with a 'Today' button and a calendar icon. A note at the bottom states 'Note: You are 8 hours ahead of server time.'

- ① Import "datetime" module
- ② Add field using **DateField** type
- ③ Add default option using **today** function

For database management, you may want to record the data registration date. The easiest way to set a **timestamp** is using the `auto_now_add` argument in `DateField`. However, when you use `auto_now_argument=True`, the field becomes uneditable and is not shown on the Django admin page. On this lesson, we'll explain how to make a date timestamp which is editable and is shown in the Django admin page.

There are three points to implement it.

- **Import the datetime module:** To use the `today` function, you need to import the `datetime` module first.

- **Add a field using DateField type:** To store the date in date format, use `DateField`.
- **Add the default option using the today function:** Add the `default` option with the `today` function as the default value

Practice

Objective:

Add an editable data registration date

In this practice, we'll explain how to add the data registration date as one of the date fields.

1. Edit models.py

Edit the `Employee` model in `models.py`, covering the following points:

- Import the `datetime` module
- Add a field using the `DateField` type
- Add the default option using the `today` function
- Add the `verbose_name` option with "Data Registration Date" as a value. This is used for making the data field name more explicit in browsers.

The yellow line is the additional code.

`employee_learning/models.py`

```
from django.db import models
import datetime

class Employee(models.Model):
    PRIORITIES=[('H', 'High'), ('M', 'Medium'),
    ('L', 'Low'), ]
```

```
name=models.CharField(max_length=25,  
verbose_name="Employee Name")  
priority=models.CharField(max_length=1,  
choices=PRIORITIES, default="M",  
verbose_name="Learning Priorities")  
reg_date=models.DateField(default=  
datetime.date.today, verbose_name="Data  
Registration Date")  
  
def __str__(self):  
    return self.name
```

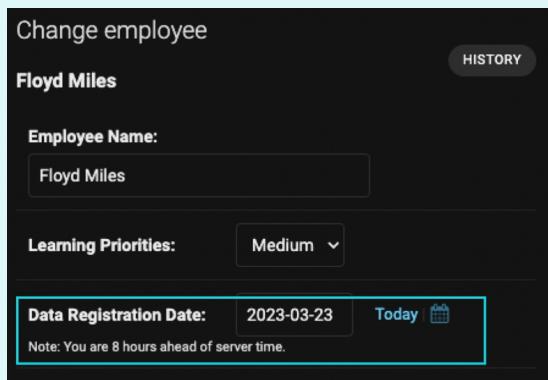
2. Execute the changes in the database and check the admin site

Run the three usual commands and go to the Django admin site.

Command Line - INPUT

```
(d_env) project_d % | python manage.py  
makemigrations employee_learning  
(d_env) project_d % | python manage.py migrate  
(d_env) project_d % | python manage.py runserver
```

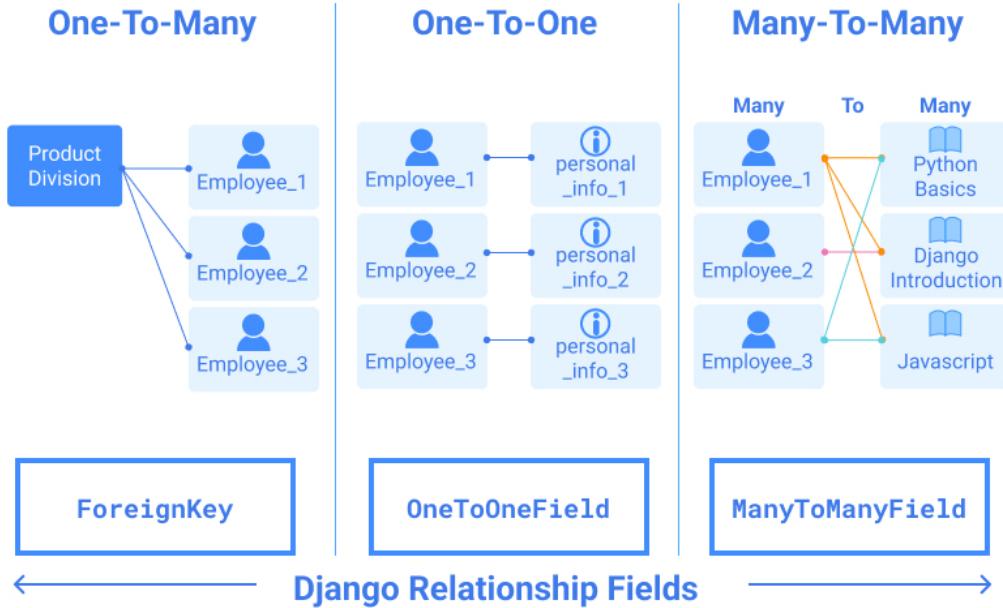
You can confirm that the data registration date is reflected in the [Employee](#) model. After this step, the date will be amended to the data registration date.



Note: When you add this date field to an existing model with data records, the data entry time will be equivalent to the time you added the date field. For

the existing records, you need to adjust the date field manually.

Django Models – Relationship Fields



As explained at the beginning of this chapter, the relational database has three types of relationships. Django handles the relationships using **Relationship Fields**.

One-To-Many relationship: ForeignKey

The **One-To-Many** relationship is the most frequently used relationship in the relational database. In this relationship, each record in one table can correspond to many records in another table, but not the other way around. In Django, this relationship is handled through the `ForeignKey` data field.

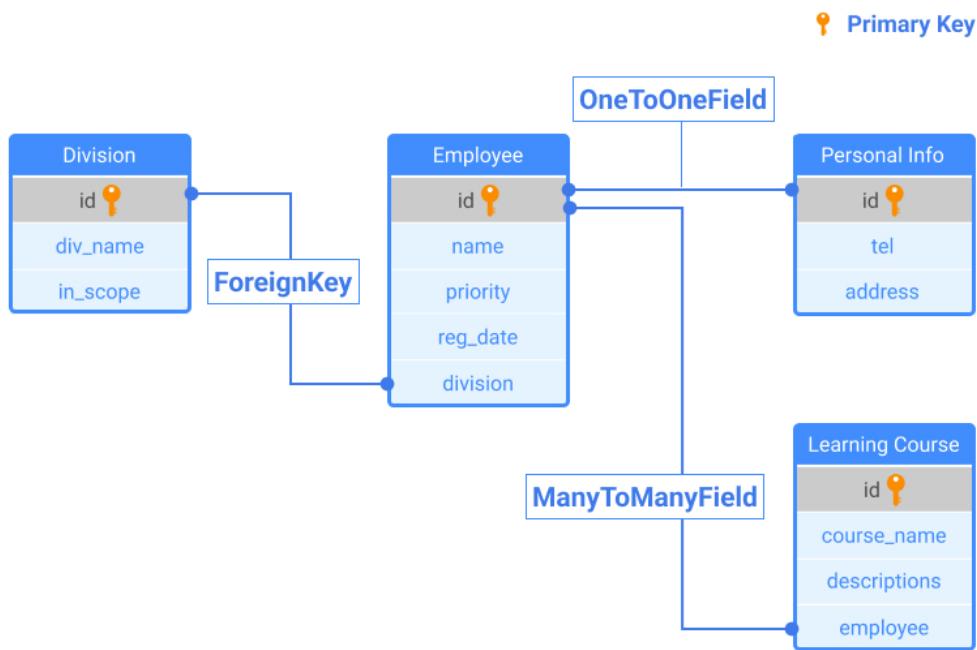
One-To-One relationship: OneToOneField

In the **One-To-One** relationship, each record in one table corresponds to one record in another table. In Django, this relationship is handled through `OneToOneField`. One-to-one relationship tables can be combined into one table, but there are reasons to separate tables from the data management point of view.

Many-To-Many relationship: `ManyToManyField`

In the **Many-To-Many** relationship, each record in one table can correspond to many records in another table, and vice versa. Usually, a many-to-many relationship between two tables is defined through an **intermediary join table (associative entity)**. In Django, this relationship is handled through `ManyToManyField`. By connecting the two tables using `ManyToManyField`, Django automatically creates the intermediary join table.

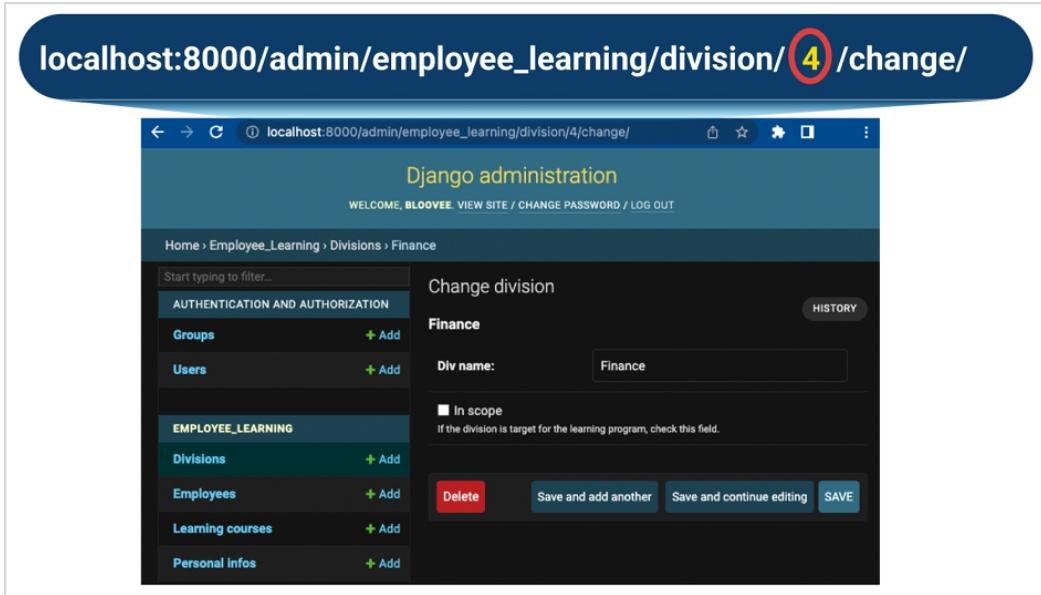
Django Models – ID



When you create a model, Django automatically adds an `id` field to each model, which is used as the **primary key** for the model. The `id` field is not shown in the Django admin site by default; however, the `id` field plays a critical role in relationships.

Check ID

Although the **primary key ID (Django ID)** is not shown directly in the Django admin site, there is a way to check the ID. When you create a record, Django admin creates a page to update the record. The ID of the record is shown as part of the URL. For example, go to the `Division` model page and click on one of the records. You can see the ID of the record.

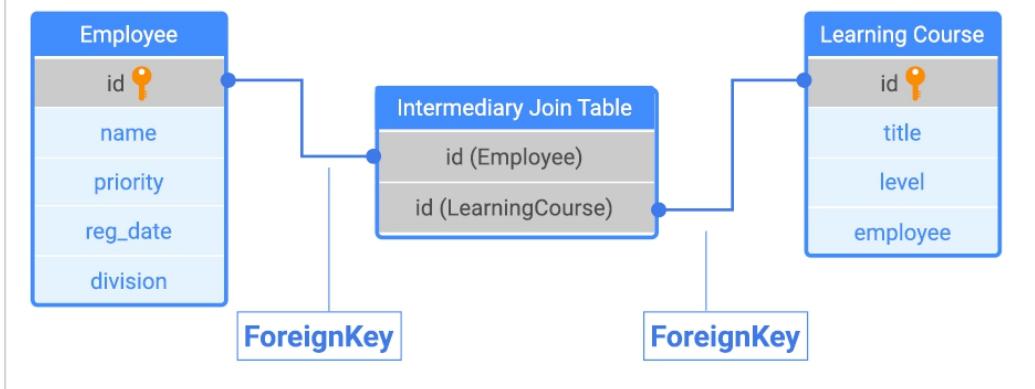


Relationships with ID

In Django, you'll define relationships from one model using relationship fields (`ForeignKey`, `OneToOneField`, or `ManyToManyField`), specifying a related model as the key argument. Then, when the relationship is defined through the relationship fields, the two tables are connected through the ID of each table.

As explained in the previous section, the situation is more complicated for the `ManyToManyField` case. Behind the scenes, Django creates an **intermediary join table** to enable the many-to-many relationship. IDs from each table are mapped through the intermediary join table, like in the illustration below.

Intermediary Join Table



Django Models – ForeignKey (OneToMany Relationship)



```
models.ForeignKey(RelatedModel, on_delete=xx)
```



To make a **One-To-Many** relationship, Django uses `ForeignKey`. There are two key arguments you need to specify.

1. Related model name
2. `On_delete` argument

As explained on the previous page, by specifying a related model name through the `ForeignKey` field, Django connects the field with the ID of the related model. The `on_delete` argument is used to define the behavior when an object of the related model is deleted.

There are three possible values for the `on_delete` argument.

- **CASCADE**: deletes the object containing the `ForeignKey`.
- **PROTECT**: prevents deletion of the referenced object by raising `ProtectedError`
- **RESTRICT**: prevents deletion of the referenced object by raising `RestrictedError`

As the possible values are available in `django.db.models`, you need to write `on_delete=models.CASCADE`.

Practice

Objective:

Learn how to use `ForeignKey`

In this practice, we'll explain how to use `ForeignKey` to create **One-To-Many Relationship**. In this case example, we'll add a `division` field in the `Employee` model by connecting it with the `Division` model.

1. Add a new field in the `Employee` model in `models.py`

Edit the `Employee` model in the `models.py` file to connect with the `Division` model.

- **Field name:** `division`
- **Field type:** `ForeignKey`
- **Related model name:** `Division`
- **On_delete:** `models.CASCADE`

The **yellow** line below is the new code.

`employee_learning/models.py`

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

```
import datetime

class Employee(models.Model):

    PRIORITIES=[('H', 'High'), ('M', 'Medium'),
    ('L', 'Low'),]

    name=models.CharField(max_length=25,
    verbose_name="Employee Name")
    priority=models.CharField(max_length=1,
    choices=PRIORITIES, default="M",
    verbose_name="Learning Priorities")
    reg_date=models.DateField(default=
    datetime.date.today, verbose_name= "Data
    Registration Date")
    division=models.ForeignKey(Division,
    on_delete=models.CASCADE)

    def __str__(self):
        return self.name
```

2. Move the Division model before the Employee model in models.py

Cut the code of the *Division* model and paste the code before the *Employee* model in the *models.py* file. This is needed because the *Employee* model will refer to the *Division* model in the new field.

employee_learning/models.py

```
from django.db import models
import datetime

class Division(models.Model):
    :
    def __str__(self):
        return self.div_name
class Employee(models.Model):
    :
```

3. Run the makemigrations command

Run the `makemigrations` command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py  
makemigrations employee_learning
```

As there are existing records, you'll encounter the message below.

Command Line - INTERACTIVE

It is impossible to add a non-nullable field 'division' to employee without specifying a default. This is because the database needs something to populate existing rows.

Please select a fix:

- 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
- 2) Quit and manually define a default value in models.py.

Select an option:

Answer [1](#) at this time. Then, the prompt is asking what data should be filled in.

Command Line - INTERACTIVE

Select an option: 1

Please enter the default value as valid Python.

The datetime and django.utils.timezone modules are available, so it is possible to provide e.g. timezone.now as a value.

Type 'exit' to exit this prompt

>>>

Here, you need to be careful. You should not input the data that is not available. As we need to input an available ID created in the *Division* model, type "[1](#)" for now.

Command Line - INTERACTIVE

```
>>> 1  
Migrations for 'employee_learning':  
    employee_learning/migrations/0007_employee_divis  
    ion.py  
        - Add field division to employee
```

4. Migrate the model into the database and run the runserver

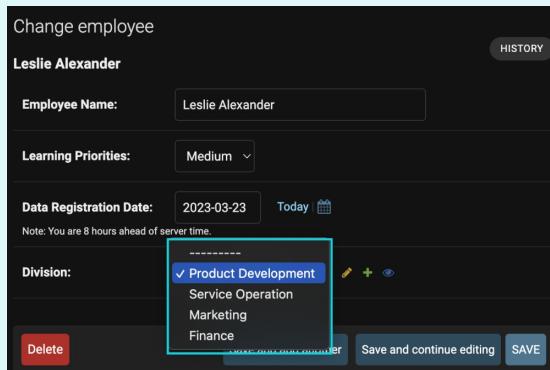
Complete the `migration` and run the `runserver` command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py migrate  
(d_env) project_d % | python manage.py runserver
```

5. Check the Django admin site

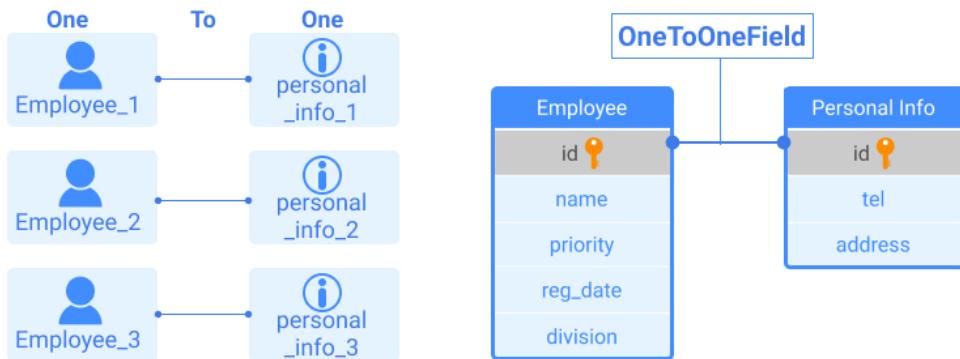
You can see that the `Division` model and the `Employee` model are connected. The employee model also reflects the data as we temporally input "1" for existing data. ID:1 is "`Product Development`" in this case.



Django Models – OneToOneField

models.py

```
models.OneToOneField(RelatedModel,  
on_delete=xx, primary_key=True)
```



To make a **One-To-One relationship**, Django uses `OneToOneField`. Just like in `ForeignKey`, there are two key arguments you need to specify.

1. Related model name
2. On_delete argument

In addition to these arguments, you may need to add the `primary_key` option for the field.

Practice

Objective:

Learn how to use OneToOneField

In this practice, we'll explain how to use `OneToOneField`. In this case example, we'll create a new model `PersonalInfo` and connect it with the `Employee` model using One-To-One relationship.

1. Add Personal Info model in models.py

Create the `PersonalInfo` model in the `models.py` file with `OneToOneField` to connect with the `Employee` model.

Add the model after the `Employee` model, as the `PersonalInfo` model refers to the `Employee` model. Below is the code for the `PersonalInfo` model.

The **yellow** lines below are the new code.

`employee_learning/models.py`

```
:  
class Employee(models.Model):  
:  
class PersonalInfo(models.Model):  
    name=models.OneToOneField(Employee,  
    on_delete=models.CASCADE, primary_key=True)  
    tel=models.CharField(max_length=15)  
    address=models.CharField(max_length=50)
```

2. Add Personal Info model in admin.py

You need to add the new model in the `admin.py` file to reflect the model.

`employee_learning/admin.py`

```
from .models import Employee  
from .models import Division  
from .models import PersonalInfo  
  
admin.site.register(Employee)
```

```
admin.site.register(Division)
admin.site.register(PersonalInfo)
```

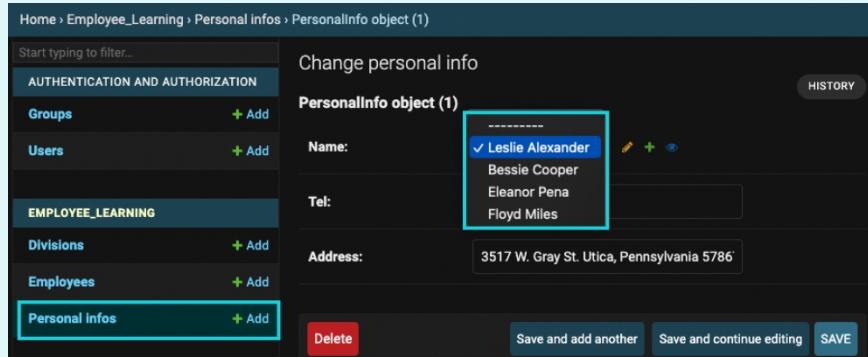
3. Execute the changes in the database and check the admin site

Run the three usual commands and go to the Django admin site.

Command Line - INPUT

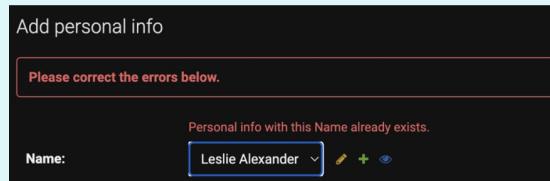
```
(d_env) project_d % | python manage.py
makemigrations employee_learning
(d_env) project_d % | python manage.py migrate
(d_env) project_d % | python manage.py runserver
```

You can see that the *PersonalInfo* model is added and connected with the *Employee* model.



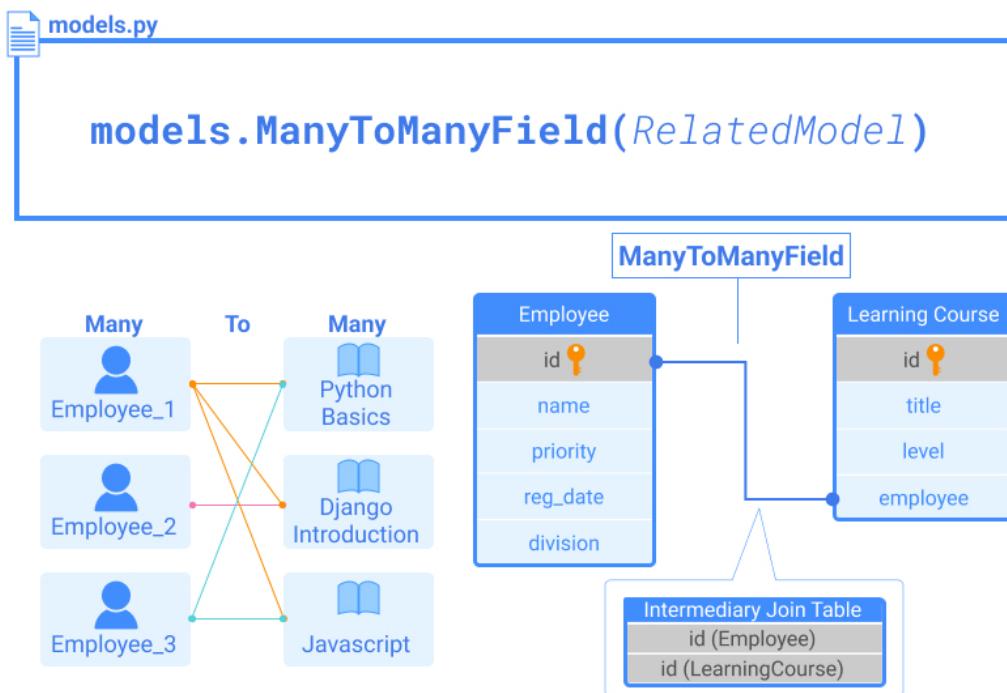
The screenshot shows the Django Admin interface for the 'Personal infos' model. The left sidebar has 'Personal infos' highlighted with a red border. The main area shows a table with one row for 'Leslie Alexander'. The 'Name' column contains a dropdown menu with options: 'Leslie Alexander' (selected), 'Bessie Cooper', 'Eleanor Pena', and 'Floyd Miles'. The 'Address' column shows '3517 W. Gray St. Utica, Pennsylvania 5786'. At the bottom are buttons for 'Delete', 'Save and add another', 'Save and continue editing', and 'SAVE'.

As we set the **primary key** for the OneToOne relationship field, you'll encounter an error message when you try to register personal info for the same employee twice.



The screenshot shows the 'Add personal info' form. A red box highlights the error message 'Please correct the errors below.' Below it, another red box highlights the error 'Personal info with this Name already exists.' The 'Name' field contains 'Leslie Alexander' with a dropdown arrow. At the bottom are buttons for 'Save and add another', 'Save and continue editing', and 'SAVE'.

Django Models – ManyToManyField



To make a **Many-To-Many relationship**, Django uses `ManyToManyField`. For the field, you need to specify a **related model name**. To connect the two tables using `ManyToManyField`, Django automatically creates the **intermediary join table**.

Practice

Objective:

Learn how to use `ManyToManyField`

In this case example, we'll create a new model `LearningCourse`, and connect it with the `Employee` model using a Many-To-Many relationship.

1. Add LearningCourse model in models.py

Create the *LearningCourse* model in the *models.py* file with *ManyToManyField* to connect with the *Employee* model.

Add the model after the *Employee* model, as the *LearningCourse* model refers to the *Employee* model. Below is the code for the *LearningCourse* model.

The **yellow** lines below are the new code.

employee_learning/models.py

```
:  
class Employee(models.Model):  
:  
class PersonalInfo(models.Model):  
:  
class LearningCourse (models.Model):  
  
    LEVEL=[('B','Basic'), ('I','Intermediate'),  
    ('A','Advanced')]  
    title = models.CharField(max_length=50,  
    unique=True)  
    level = models.CharField(max_length=1,  
    choices=LEVEL)  
    employee = models.ManyToManyField(Employee)  
  
    def __str__(self):  
        return self.title
```

2. Add LearningCourse model in admin.py

You need to add the new model in the *admin.py* file to reflect the model.

employee_learning/admin.py

```
from .models import Employee  
from .models import Division  
from .models import PersonalInfo  
from .models import LearningCourse  
  
admin.site.register(Employee)
```

```
admin.site.register(Division)
admin.site.register(PersonalInfo)
admin.site.register(LearningCourse)
```

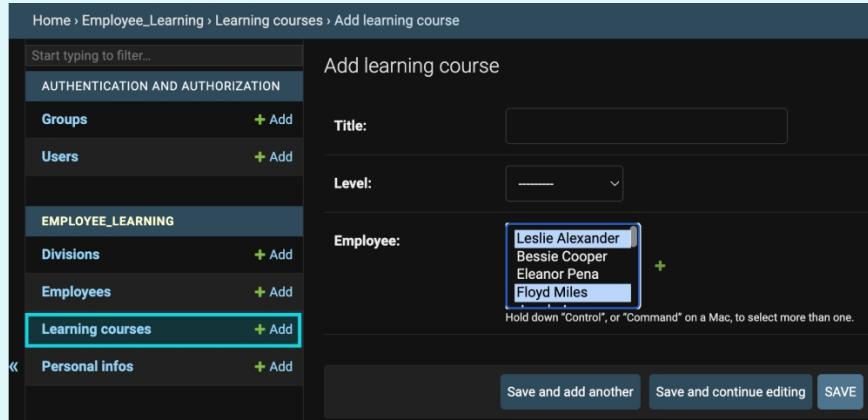
3. Execute the changes in the database and check the admin site

Run the three usual commands and go to the Django admin site.

Command Line - INPUT

```
(d_env) project_d % | python manage.py
makemigrations employee_learning
(d_env) project_d % | python manage.py migrate
(d_env) project_d % | python manage.py runserver
```

You can see that the *LearningCourse* model is added and connected with the *Employee* model. As the relationship is Many-To-Many, you can select multiple employees.



Chapter 4

Create CRUD Web Application

This chapter covers how to make a basic CRUD web application using the Django framework.

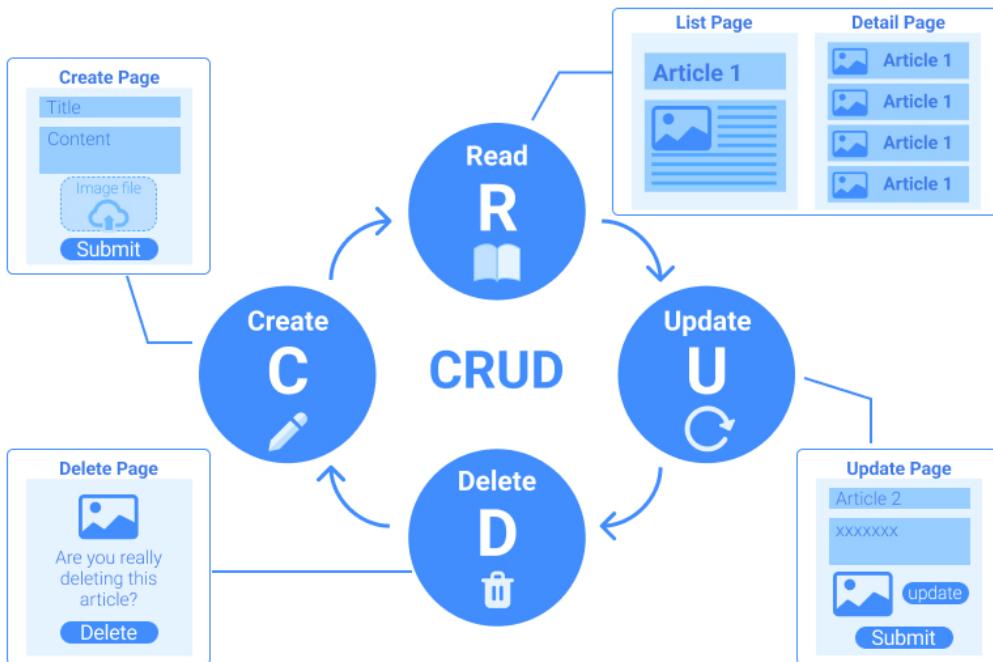
The following topics are covered in this chapter.

Topics

1. [CRUD Web Application](#)
2. [Basic CRUD Structure in Django](#)
3. [Django Generic Views](#)
4. [How To Write Class-Based Views with Generic Views](#)
5. [Generic View Basic Attributes](#)
6. [URL Dispatcher for CRUD Views](#)
7. [Django Templates for CRUD Views](#)
8. [Django Template Language \(DTL\)](#)
9. [Template for List Page](#)
10. [get_FOO_display method](#)
11. [Template for Detail Page](#)

- [**12. Template with Model Relations**](#)
- [**13. Template for Create and Update Page**](#)
- [**14. Template for Delete Page**](#)
- [**15. Add Links – {url %} tag**](#)
- [**16. Extend Templates – {extends %} tag**](#)
- [**17. Check Developing App UI on Mobile Device**](#)
- [**18. Django Templates with Bootstrap**](#)
- [**19. Crispy Forms**](#)
- [**20. Customize Views \(1\) – Change List Order**](#)
- [**21. Customizing Views \(2\) – Filter Lists**](#)
- [**22. Context**](#)
- [**23. Customize Views \(3\) – Add Extra Context**](#)
- [**24. Modularize Templates – {include %} tag**](#)
- [**25. Static Files in Development Environment – {static %} tag**](#)
- [**26. STATIC_URL and STATICFILES_DIRS**](#)
- [**27. Create Index HTML**](#)

CRUD Web Application



CRUD is the acronym for **Create**, **Read**, **Update**, and **Delete**. The four functions represent the minimum operations used in database applications.

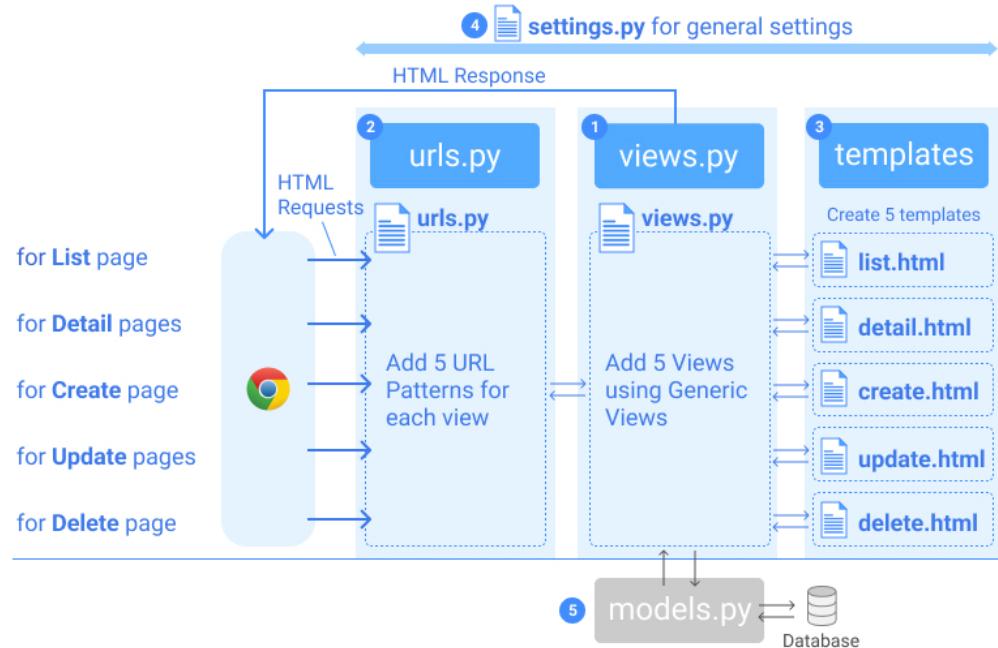
- **Create (C):** Creates new data records in the system. For example, in blog applications, bloggers post their blog articles, and the system registers the posted article.
- **Read (R):** Read (retrieves) existing data records in the system. For example, the system retrieves posted articles to display them.
- **Update (U):** Updates existing data records. For example, bloggers update their posted blog articles, and the system updates the records.
- **Delete (D):** Deletes existing data records from the system. For example, bloggers delete their posted blog articles, and the system removes the records.

CRUD in web applications

In web applications, the CRUD concept is executed through web pages. For example:

- Content editors go to the content posting page (*Create Page*) to post their content.
- The posted content can be read by users on a list-style page with a list of many content items (*List Page*) or a particular content page (*Detail Page*).
- When content editors want to edit or delete the existing content, they can edit or delete it on the *Update Page* or *Delete Page*.

Basic CRUD Structure in Django



Django framework is well-designed to execute the **CRUD concept** efficiently. Five key design points exist to build a CRUD-based web application in Django.

1 views.py

`views.py` is used to build logic to handle HTTP requests dispatched from `urls.py`. You can build the logic from scratch; however, Django provides pre-made logic templates named **generic views**. By using generic views, you can build web applications with simple code. To handle each CRUD page, you need to design different views for each page.

2 urls.py

urls.py is used as a **URL dispatcher**. HTTP requests from a browser are handled by the URL dispatcher first. It dispatches the requests to related views (defined in the *views.py* file). To handle each CRUD page, you need to define different URL patterns for each page.

3 Templates

Django templates usually consist of multiple HTML files. They are saved under the *templates* directory, whose path is defined in *settings.py*. The template HTML files are used to convert data handled by *views.py* into the HTML format. To handle each CRUD page, you usually need to prepare different HTML files for each page.

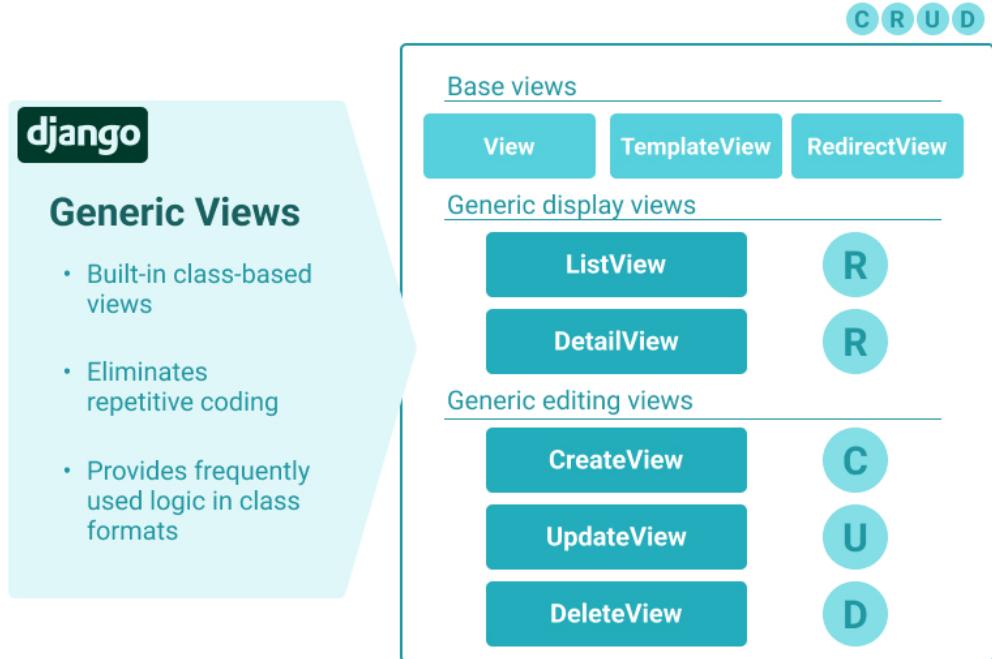
4 settings.py

settings.py is used to define general settings for Django applications, such as defining key files or directory paths or registering domains to allow access to the web application.

5 models.py

models.py is used to design a database for the web application. As we already explained in the previous chapter, we don't cover Django models in this chapter; however, the case examples in this chapter use the model designed in the previous chapter.

Django Generic Views



Generic views in Django are built-in **class-based views**. Generic views are developed to eliminate repetitive coding by providing frequently used logic in class format. Using generic views, you can build web applications with simple code.

Different types of generic views are used depending on the objectives. In Chapter 2, we explained *TemplateView*, which is categorized under Base views. **Base views** also include *View*, which is the ancestor of all other class-based views, and *RedirectView*, which is used for URL redirections.

In a simple CRUD application, there are five frequently used generic views.

1. *ListView*

2. *DetailView*
3. *CreateView*
4. *UpdateView*
5. *DeleteView*

They are categorized into two groups depending on their purposes – **generic display views** and **generic editing views**.

Generic display views

Generic display views are designed to display data. There are two views under this category – *ListView* and *DetailView*.

[ListView](#)

ListView is used to provide logic to display a list of objects. For example, *ListView* retrieves a list of blog articles from a database in the blog application.

[DetailView](#)

DetailView is used to provide logic to display a particular object. For example, the blog application retrieves a particular blog article from a database.

Generic editing views

Generic editing views are designed to provide a foundation for editing content. There are four views under this category – *FormView*, *CreateView*, *UpdateView*, and *DeleteView*. In this Chapter, we'll cover the following three views.

[CreateView](#)

CreateView is used to provide logic to display an input form to create an object. When inputs are submitted, *CreateView* handles the inputs and registers the data in a database.

UpdateView

UpdateView is used to provide logic to display an input form to update an object. When inputs are submitted, *UpdateView* handles the modification of the registered data in a database.

DeleteView

DeleteView is used to provide logic to display a page for the data delete action and handle the data delete process.

How To Write Class-Based Views with Generic Views

```
views.py
from django.views.generic import ListView
from .models import MyModel
class MyView(ListView):
    template_name = 'list.html'
    model = MyModel
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

The code snippet shows a Python file named `views.py`. It contains a class-based view `MyView` that inherits from `ListView`. The code is annotated with four numbered callouts:

- ① `from django.views.generic import ListView` and `from .models import MyModel`
- ② `class MyView(ListView):`
- ③ `template_name = 'list.html'` and `model = MyModel`
- ④ `def get_context_data(self, **kwargs):`, `context = super().get_context_data(**kwargs)`, `context['now'] = timezone.now()`, and `return context`

- ➊ Import generic views and models used in this view
- ➋ Create a new view using a generic view
- ➌ Set 'attribute values' for each attribute defined for each generic view
- ➍ Customize the generic view (e.g., set a new context)

Django generic views are designed using **Python classes**. How to use generic views in `views.py` follows the **Python class syntax**. Basically, you'll need to create a new view as a child class of a built-in generic view. The new view inherits the attributes and methods from the built-in generic view. When you make a new view using a generic view, the code has four sections.

- ➊ Import generic views and models used in this view
- ➋ Create a new view using a generic view
- ➌ Set 'attribute values' for each attribute defined for each generic view

4. Customize the generic view (e.g., set a new context)

1 Import generic views and models used in this view

Import generic views

First, you need to import the generic views you want to use. Generic views are stored in `django.views.generic`. In the example in the main figure, `ListView` is imported.

Import models

You also need to import models you want to use in your view. The `.` (dot) means that the same directory as the `views.py` is located. In the example in the main figure, `MyModel` is imported.

2 Create a new view using a generic view

As explained in Chapter 2, you can create a view through inheritance.

The way to write a class-based view follows Python class syntax.

- Class name is usually written in **PascalCase** (Capitalize all words) like `MyView` in the main figure.
- Use the **parent class name** as an argument of the class to inherit the parent class, like `ListView` in the main figure

3 Set 'attribute values' for each attribute defined for each generic view

Each generic view has its own attributes. As generic views inherit attributes from their ancestor views, several common attributes are used for different generic views. There are two attributes in the main figure example. There are many other attributes. We'll explain the key ones in the next section.

- `template_name`: set an html file used for this view

- `model`: set a model to be used for this view

4 Customize the generic view

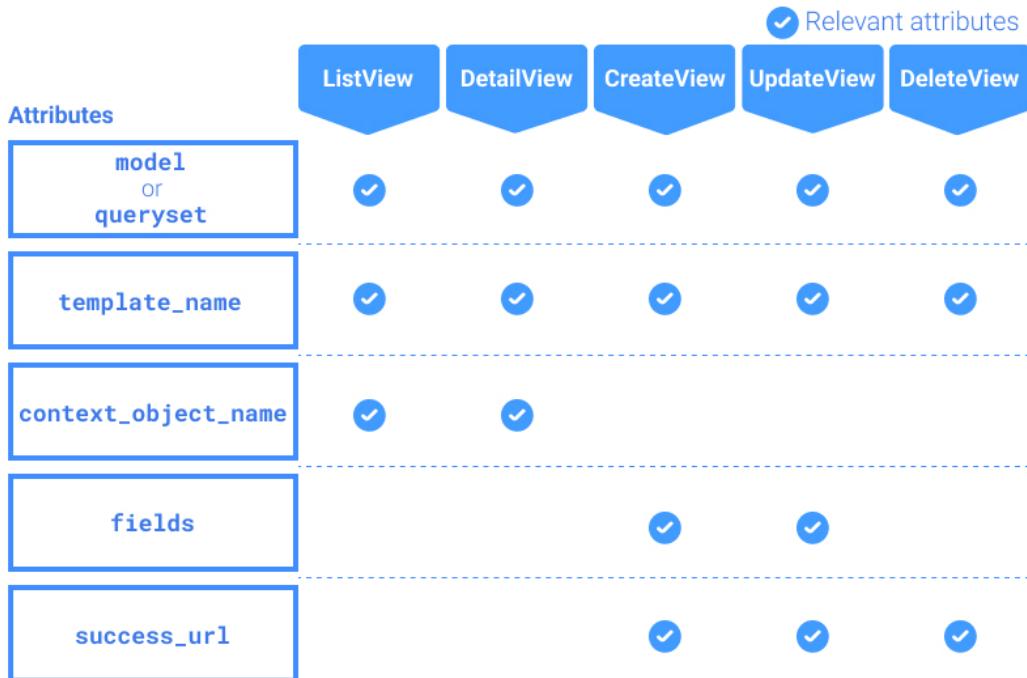
Although generic views are equipped with basic functionalities, you may need to customize them. For example, you can do the following things by customizing generic views.

- add a query to show selected data or to change the order of content items
- add context (a dictionary of data that is passed from a view to an HTML template)
- add logic to display content such as differentiating user interface depending on login status

Customization of views is done in a way that overrides the method defined in the generic view. In the main figure example, `get_context_data` is a method to get the **context data** of the model. The `get_context_data` method is accessible from `ListView`, but it is initially defined in the `SingleObjectMixin` class. `ListView` inherits the method from the `SingleObjectMixin` class.

We'll explain the **context** and how to override a method later in this chapter.

Generic View Basic Attributes



Attributes in generic views are key design elements when you create a new view using generic views. For example, you can define which template file will be used for the view or which model will be used for the view. Required or optional attributes are different in each generic view.

 **views.py**

```
class MyView(ListView):
    template_name = xxx
    model = xxx
```

Attribute

Basic attributes frequently used in generic views

There are many attributes you can use when creating new views with generic views. The following are basic frequently used attributes:

model

The `model` attributes define the model for which the view will display data. The `model` attribute can be replaced by the `queryset` attribute. The `model` attribute is defined in *SingleObjectMixin* and *MultipleObjectMixin*.

queryset

The `queryset` attribute is also used to define the model used for the view; however, `queryset` gives you more functionalities using several methods.

- `all()`: retrieves all objects. '`queryset = ModelName.objects.all()`' give the same result as '`model = ModelName`'
- `filter()`: returns objects that match the given lookup parameters
- `order_by()`: returns objects in order. To reverse the order, you can use `"-"`. For example, if you want to order based on the `name` field, use `order_by('name')`. If you want to reverse the order, you can use `order_by('-name')`.

The `queryset` attribute is defined in *SingleObjectMixin* and *MultipleObjectMixin*.

template_name

The `template_name` attribute is used to define the path of an HTML template for the view. The path should be described as a relative path from the template directory set in the `settings.py` file.

For example, when you set the '`templates`' directory in `settings.py` as the main directory for the templates and directly place the template file named `template.html` in the '`templates`' directory, you can just set it like '`template_name = template.html`'.

When you want to use a subdirectory, use the subdirectory name before the template file name, like '`template_name = subdir/template.html`'.

If you don't specify this attribute, Django uses a predefined path for the template.

For example, if an app name is `employee_learning`, a model name is `LearningCourse`, and a generic view is `ListView`, the inferred HTML template will be `employee_learning/learningcourse_list.html`.

To organize template files better, specifying `template_name` is recommended.

The `template_name` attribute is defined in `TemplateResponseMixin`.

context_object_name

The `context_object_name` attribute is used to add a name to the object passed to template files.

For example, the default `object` name is `object` in `DetailView`. In `DetailView`, `object_list` contains the list of objects. You can call objects defined in related views using '`object`' or '`object_list`' in Django templates.

When you specify `context_object_name` in your view, you can call the objects using the name you defined.

Even after specifying `context_object_name`, you can still call the objects using '`object`' or '`object_list`'. The `context_object_name` attribute gives you an additional name that can be used in your template files.

The `context_object_name` attribute is defined in `SingleObjectMixin` and `MultipleObjectMixin`.

We'll explain about `context_object_name` in detail later in this chapter.

fields

The `fields` attribute is used to define model fields shown in the input form pages (create page and update page). This attribute is required if you are generating the form class automatically.

The `fields` attribute is defined in *ModelFormMixin*.

success_url

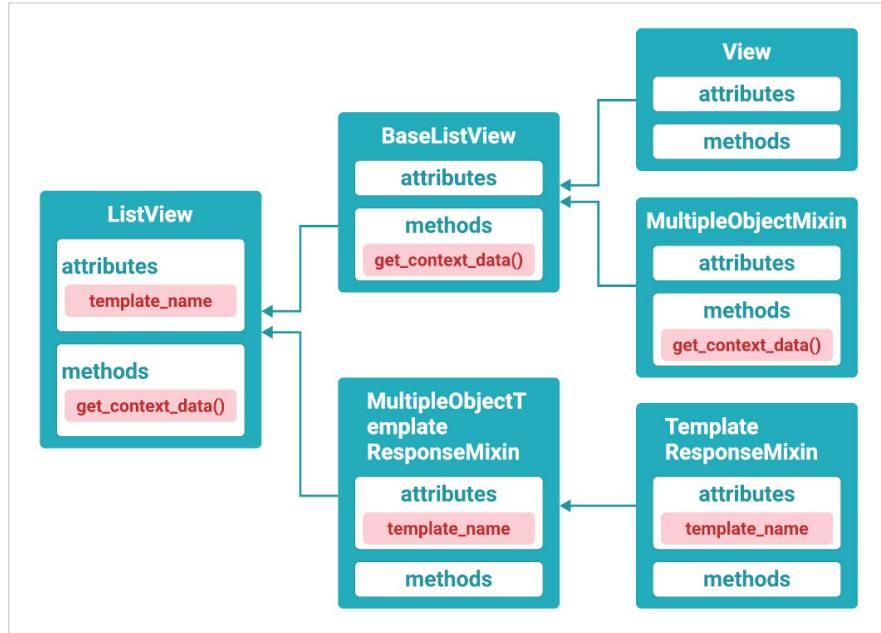
The `success_url` is used to define the URL to redirect to when the form is successfully processed. This attribute is usually required for generic editing views.

`reverse_lazy()` is often used for this argument to resolve Django URL names into URL paths. Django URL names are defined in `urls.py`, which will be explained later.

The `fields` attribute is defined in *FormMixin*.

Note: Generic Views Inheritance

Each generic view has attributes and methods that are inherited from their ancestor views or mixins (a set of methods or attributes that can be "mixed in" to a class).



In the Django documentation, you may not be able to find attribute information under each generic view section.

Attribute or method information is written in the ancestor classes (views or mixins) in the Django documentation.

[Django documentation reference: ListView](#)

[Django documentation reference: MultipleObjectMixin](#)

[Django documentation reference: TemplateResponseMixin](#)

Practice

In this practice, we'll create views using Django generic views. We also use the `LearningCourse` model we designed in the previous chapter.

1. Import generic views and models used in this view

All generic views are stored in `django.views.generic`.
Import five generic views

- `ListView`
- `DetailView`
- `CreateView`
- `UpdateView`
- `DeleteView`

Also, import the `LearningCourse` model.

The **yellow** lines below are the new code.

`employee_learning/views.py`

```
from django.shortcuts import render
from django.views.generic import ListView, DetailView,
CreateView, UpdateView, DeleteView
from .models import LearningCourse
```

2. Create the CourseList and CourseDetail views

As a simple implementation, you set the following three attributes for both views.

- `model`: You can also use `queryset` instead of `model`.
- `template_name`: We haven't created HTML templates, but let's assume we'll create the `employee_learning` directory to store HTML templates.
- `context_object_name`: '`course_object_list`' and '`course_object`' will be used later in the template section of this chapter.

`employee_learning/views.py`

```
from django.shortcuts import render
from django.views.generic import ListView, DetailView,
CreateView, UpdateView, DeleteView
from .models import LearningCourse

class CourseList(ListView):
```

```
model = LearningCourse
template_name =
'employee_learning/course_list.html'
context_object_name = 'course_object_list'

class CourseDetail(DetailView):
    model=LearningCourse
    template_name =
'employee_learning/course_detail.html'
    context_object_name = 'course_object'
```

3. Create the CourseCreate, CourseUpdate and CourseDelete views

As `CreateView`, `UpdateView`, and `DeleteView` are generic editing views, the required attributes differ from those for `ListView` and `DetailView`. You don't need `context_object_name`, but you need to add `success_url` to define the URL to redirect when the data submission action is successful. For `CreateView` and `UpdateView`, you also need to add the `fields` attribute to specify which field should be shown in the input form.

As we use `reverse_lazy` for `success_url`, we need to import it first.

employee_learning/views.py

```
:
from .models import LearningCourse
from django.urls import reverse_lazy
:
class CourseDetail(DetailView):
    model=LearningCourse
    template_name =
'employee_learning/course_detail.html'
    context_object_name = 'course_object'

class CourseCreate(CreateView):
    model=LearningCourse
    template_name =
'employee_learning/course_create.html'
    fields=('title', 'level', 'employee')
    success_url=reverse_lazy ('course_list')

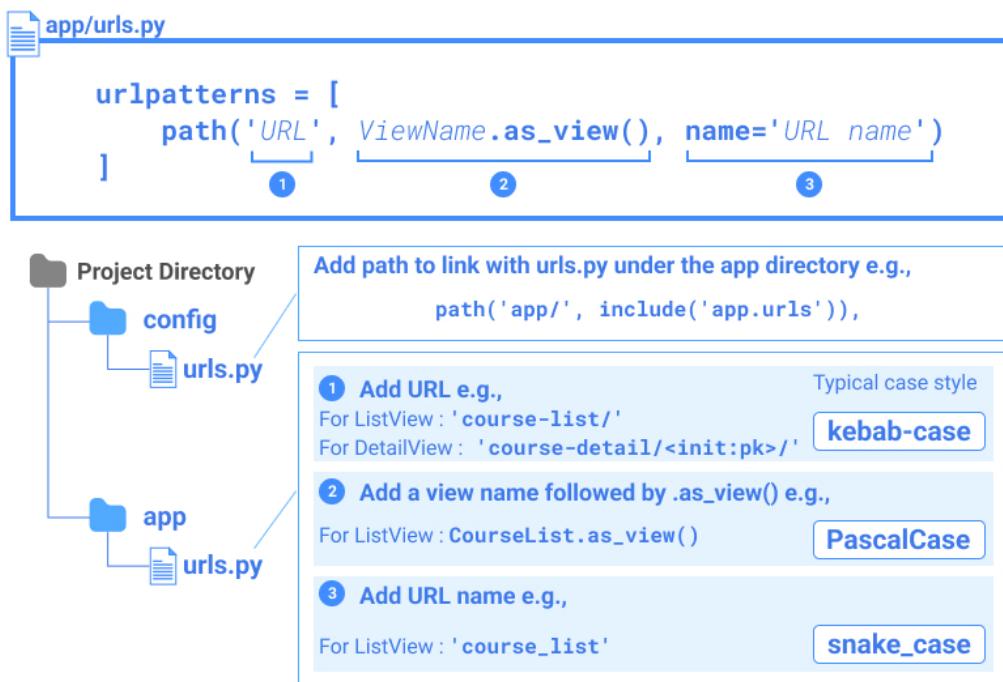
class CourseUpdate(UpdateView):
    model=LearningCourse
    template_name =
'employee_learning/course_update.html'
    fields=('title', 'level', 'employee')
```

```
success_url=reverse_lazy ('course_list')

class CourseDelete(DeleteView):
    model=LearningCourse
    template_name =
        'employee_learning/course_delete.html'
    success_url=reverse_lazy ('course_list')
```

To see the output of these views, you need to create urls.py and Django templates, which will be explained in the following sections.

URL Dispatcher for CRUD Views



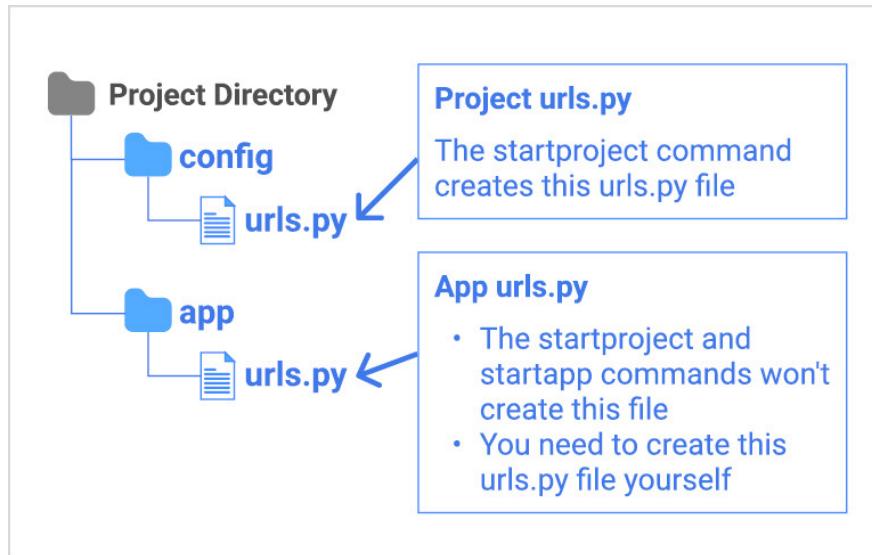
As we explained in Chapter 2, The **URL dispatcher** is one of the key concepts in the Django architecture. The URL dispatcher maps HTTP requests to the **view functions** or **classes** specified in the URL patterns written in the `urls.py` file.

In the case on the previous page, as there are five basic CRUD views, you need to prepare five **URL patterns**. Usually, application-specific URL patterns are written in the `urls.py` file in each app directory.

1. Two `urls.py` files

There are two `urls.py` files you need to handle:

- **Project urls.py:** This file is created by the `startproject` command
- **App urls.py:** A new `urls.py` file under the app directory. This file is not created by the `startproject` and `startapp` commands. You need to create it yourself.



2. Create and Edit App urls.py

As the app `urls.py` file is not automatically created, you need to create it first. Then, edit the file with the guidance below.

Import views

First, you need to import the necessary modules. You need to import all views from `views.py`.

```
from .views import CourseList, CourseDetail,
CourseCreate, CourseUpdate, CourseDelete
```

Add the five URL patterns

You need to add one URL pattern for each view. There are three arguments when you set URL patterns.

```
urlpatterns = [
    path('URL', ViewName.as_view(), name='URL name')
]
①           ②           ③
```

1. URL

In this part, you can set a URL you want to use. To avoid confusion, use a **slug** related to the view. Usually, the **kebab-case** is used for this part. You can set a variable in a part of the URL using **path converter** `<>`, which will be explained later.

2. View name

In this part, you can specify which view you want to link to the URL. For **class-based views**, usually, **PascalCase** is used. You need to add `.as_view()` after the view name for class-based views. You don't need to add `.as_view()` when you use function-based views.

3. URL name

In Chapter 2, we only covered the first two arguments but not the `name` argument. The `name` argument is useful when you handle the **reverse URL resolution**. For example, by setting this, you can call the URL in `views.py`.

Path converter – `<int:pk>`

Using a **path converter**, you can set a variable in the URL. The variable is defined using **angle brackets** `<>`. The most frequently used one is `<int:pk>`. The `int` part is a data type. `int` means zero or any positive integer. The `pk` part is a variable. `pk` means **primary key**, which is set in the auto-generated id field by default. `<int:pk>` is just an example. You can set other

types of path converters using the `<data type:name>` format. The data type examples are:

- `str` - Matches any non-empty string
- `slug` - Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters

3. Edit the project urls.py file

Creating and editing the `urls.py` file under the app directory is not enough to complete the URL dispatcher. You need to link the newly created `urls.py` file with the project `urls.py` file by adding the `include` function, as in the example below.

```
from django.urls import path, include
:
urlpatterns = [
    path('app-name/', include('app_name.urls')),
]
```

Practice

In this practice, we'll create a URL dispatcher for the views designed in the previous practice.

1. Create a new urls.py file under the employee_learning directory

The `urls.py` file is not generated by the `startapp` command. You need to create a new `urls.py` file under the `employee_learning` directory.

Command Line - INPUT

```
(d_env) project_d | cd employee_learning
(d_env) project_d | touch urls.py
```

2. Edit the app urls.py

a. Import views

As the `views.py` and `urls.py` files are in the same directory, use `.(dot)` before `views`. The following five views need to be imported.

- `CourseList`
- `CourseDetail`
- `CourseCreate`
- `CourseUpdate`
- `CourseDelete`

The `yellow` line below is the new code.

employee_learning/urls.py

```
from django.urls import path
from .views import CourseList, CourseDetail,
CourseCreate, CourseUpdate, CourseDelete
```

b. Add URL patterns for list and create pages

URL patterns are simple as you don't need to specify which data records will be displayed for the list and create pages.

employee_learning/urls.py

```
from django.urls import path
from .views import CourseList, CourseDetail,
CourseCreate, CourseUpdate, CourseDelete

urlpatterns = [
    path('course-list/', CourseList.as_view(),
         name='course_list'),
    path('course-create/', CourseCreate.as_view(),
         name='course_create'),
]
```

c. Add URL patterns for detail, update and delete pages

You need to call specific data records for the detail, update, and delete pages. To call a specific data record, add `<int:pk>` in the URL.

`employee_learning/urls.py`

```
:  
urlpatterns = [  
    path('course-list/', CourseList.as_view(),  
        name='course_list'),  
    path('course-detail/<int:pk>/',  
        CourseDetail.as_view(), name='course_detail'),  
    path('course-create/', CourseCreate.as_view(),  
        name='course_create'),  
    path('course-update/<int:pk>/',  
        CourseUpdate.as_view(), name='course_update'),  
    path('course-delete/<int:pk>/',  
        CourseDelete.as_view(), name='course_delete'),  
]
```

3. Add an urlpattern in the project urls.py

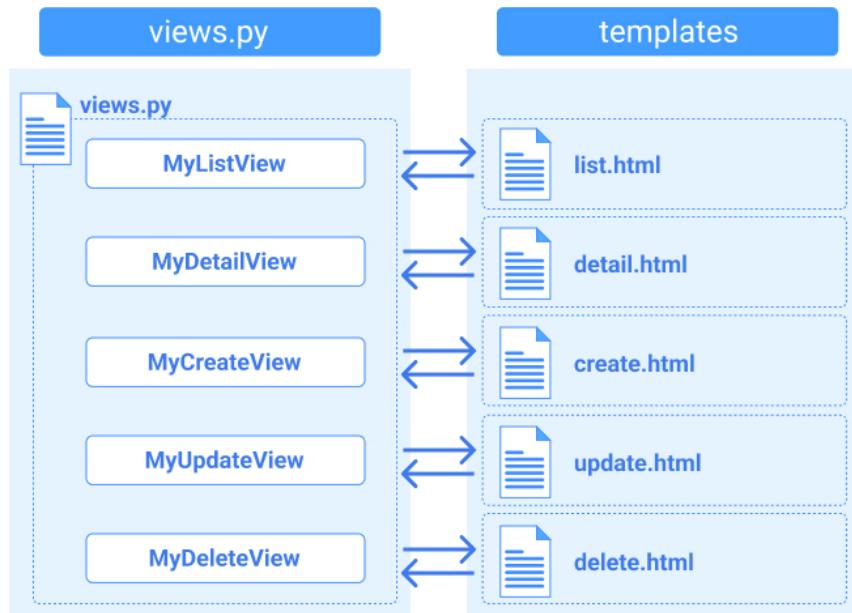
To connect paths between the two urls.py, add a urlpattern in the main urls.py under the config directory.

`config/urls.py`

```
from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('test-app/', include('test_app.urls')),  
    path('employee-learning/',  
        include('employee_learning.urls')),  
]
```

[Django documentation reference: URL dispatcher](#)

Django Templates for CRUD Views



Unlike `views.py`, you need to create different files for each view. This lesson will explain how to prepare the template html files.

Create the templates directory and register in `settings.py`

In Chapter 2, we already explained about the `templates` directory. This part is a recap of what we have already explained.

- Create the `templates` directory directly under the project directory (the `startproject` command won't create the directory)

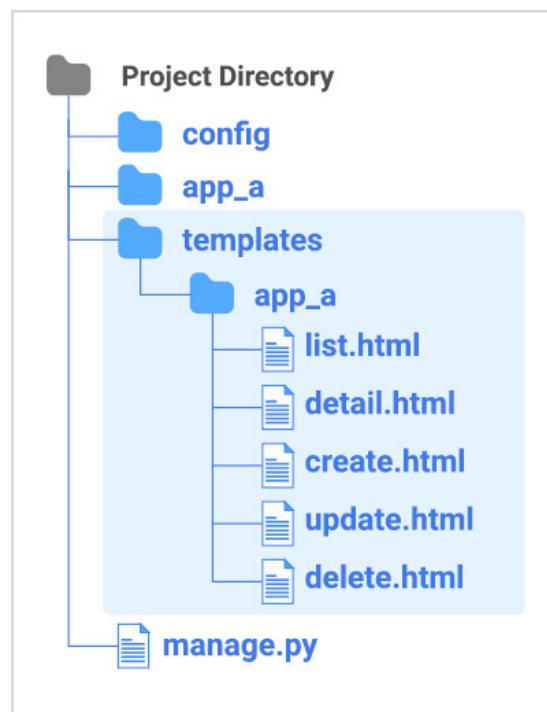
- Register the path of the *templates* directory in the *settings.py* so that Django can recognize the location of template files

Create a subdirectory for the app

As you may need HTML template files for other apps under the same project, it is better to create a subdirectory for the app-specific template files.

Create HTML templates for each view

Create HTML files for each generic view. The illustration below is an example of template files and directory structure – five HTML files are under the *app_a* directory.



Practice

In this practice, we'll create five HTML files for each of the following views created in a previous section.

- *CourseList*
- *CourseDetail*
- *CourseCreate*
- *CourseUpdate*
- *CourseDelete*

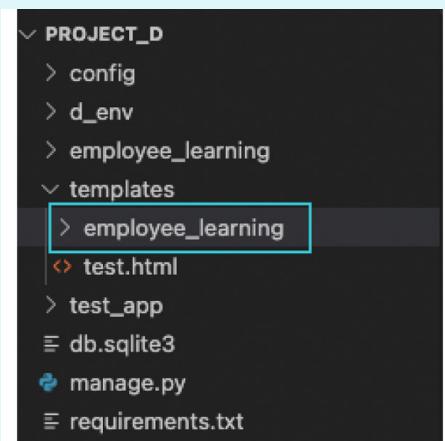
1. Create a subdirectory named *employee_learning*

We already created the *templates* directory in Chapter 2. Create a subdirectory named *employee_learning* under the *templates* directory. You can create a subdirectory with a right-click on the *templates* directory in VS code. Or go to the *templates* directory in your terminal and run the `mkdir` command.

Command Line - INPUT

```
(d_env) templates % | mkdir employee_learning
```

The directory structure will be like the one below.

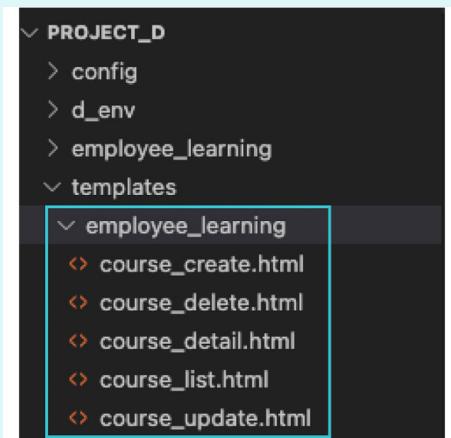


2. Create five HTML files under the *employee_learning* directory

Create the following five HTML files

- *course_list.html*
- *course_detail.html*
- *course_create.html*
- *course_update.html*
- *course_delete.html*

The directory and file structure will be like the one below.



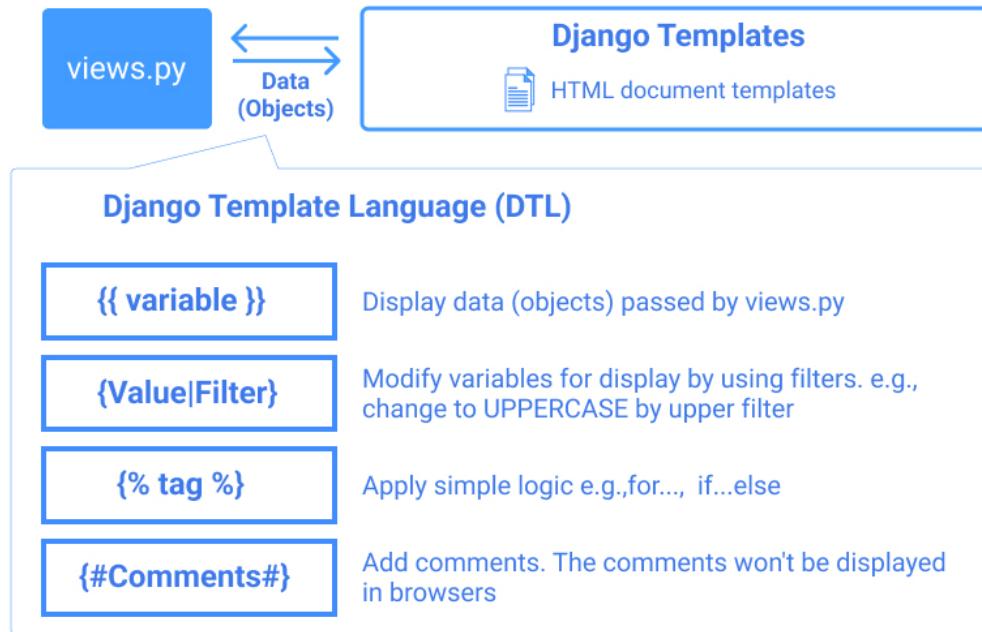
3. Edit each file

Add `<h1> Page Name </h1>` to prepare for the practices in the following sections. For example, `<h1>LIST Page</h1>` in *course_list.html*.

`templates/employee_learning/course_list.html`

`<h1>LIST Page</h1>`

Django Template Language (DTL)



Understanding the **Django Template Language (DTL)** is important to create **Django templates**. It serves as a bridge between HTML and Python code, especially for `views.py`. Using DTL, you can embed variables passed by `views.py` or add simple logic in an HTML template file.

`{{ variable }}`

You can display data (objects) passed by `views.py` using this format.

`{{ name|filter }}`

Using the filter, you can modify variables for display. For example:

- `{{ name|lower }}` displays the value of the `{{ name }}` variable in lowercase text
- `{{ name|truncatewords:n }}` displays the first n words of the `{{ name }}` variable
- `{{ text|linebreaks }}` converts line breaks in `{{ text }}` variable to `<p>` tag
- `{{ name|default:"default data" }}` displays when the `{{ name }}` variable is false or empty

{% tag %}

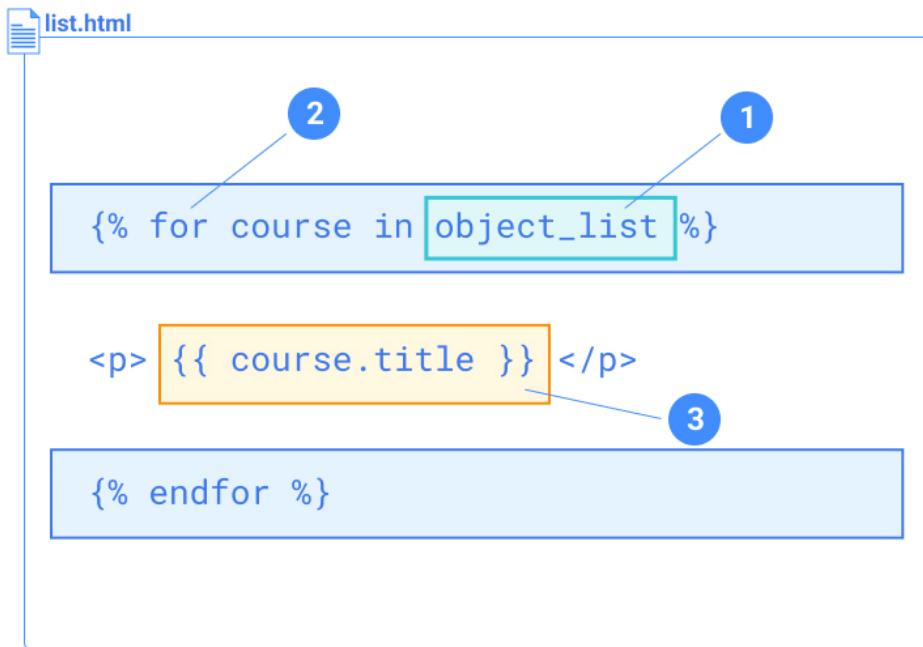
Using this format, you can apply simple logic e.g., `for...`, `if...else.`

{#Comments#}

Using this format, you can add comments. The comments won't be displayed in browsers.

[Django documentation reference: The Django template language](#)

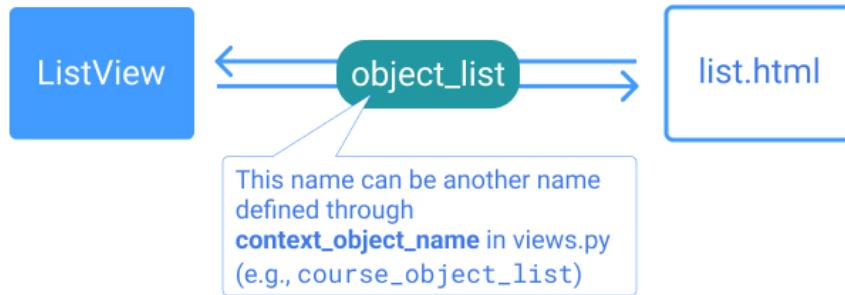
Template for List Page



The HTML template for the list page is used to display a list of items processed by `ListView`. There are three essential parts that you need to understand to make a template for `ListView`.

1. `object_list`

`object_list` is a predefined attribute by Django. It is used to bridge between `ListView` and a template (e.g., `list.html`).



You can use another name for `object_list` to make it more intuitive. When you design the view, the additional name can be set through the `context_object_name` attribute. In the case of `CourseList` that we defined earlier ([Generic View Basic Attributes](#)), `course_object_list` is the name for the list data provided by the `LearningCourse` model.



You can add another name for `object_list` to make it more intuitive. When you design the view, the additional name can be set through the `context_object_name` attribute.

2. `{% for ... in object_list %} ... {% endfor %}`

As explained on the previous page, you can add simple logic in Django templates using the [Django Template Language](#). When we make a template for `ListView`, we usually use the `for` statement.

In the main figure example, the `course` is a variable for the `for` statement. It can be any string. In this case, the part sandwiched between `{% for ... in object_list %}` and `{% endfor %}` is iterated.

3. Object attribute to be displayed

The `{{ course.name }}` part defines the actual data to be displayed. The '`course`' is a variable defined in the `for` statement. Due to the `for` statement, each record object in the table is processed one by one. The `.title` part is an attribute of '`course`' that specifies which data field (or data column) of the table is displayed.



Practice

In this practice, we'll create a simple template for `ListView` by showing only one data field – the title of the learning course. Also, we'll explain how to change the title text to Uppercase using the `filter` in this practice. In the end, we test a customized `object_list` name defined in a previous section.

1. Edit `course_list.html`

Open the `course_list.html` file, add the **yellow** lines below, and save the file.

`templates/employee_learning/course_list.html`

```
<h1>LIST Page</h1>
{% for course in object_list %}
<p> {{ course.title }}</p>
<hr>
{% endfor %}
```

2. Check the result in a browser

After saving the file, execute the `runserver` command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py runserver
```

Go to `localhost:8000/employee-learning/course-list/`

The URL is defined by the two `urls.py` files on the topic page [URL Dispatcher for CRUD Views](#). You can see that the list of course titles is displayed.

LIST Page

Web Application Basics

HTML & CSS Introduction

HTML & CSS Advanced

Linux OS Introduction

You might not have added dummy data yet if no data is displayed. Go to the Django admin page (`localhost:8000/admin/`) and add dummy data for the `LearningCourse` model.

3. Add the uppercase filter

Adding the `uppercase` filter like the one below allows you to change the title name to uppercase.

`templates/employee_learning/course_list.html`

```
:
```

```
{% for course in object_list %}
```

```
<p> {{ course.title|upper }}</p>
```

```
<hr>
```

```
{% endfor %}
```

After saving the HTML file, recheck the site. You'll see that the list is shown in uppercase.

LIST Page

WEB APPLICATION BASICS

HTML & CSS INTRODUCTION

HTML & CSS ADVANCED

LINUX OS INTRODUCTION

4. Check the user-friendly name for `object_list`

In `CourseList`, we set `context_object_name = 'course_object_list'`. Check if it works by updating the code like the yellow code below.

`templates/employee_learning/course_list.html`

```
:
```

```
{% for course in course_object_list %}
```

```
<p> {{ course.title|upper }}</p>
```

```
:
```

Go to `localhost:8000/employee-learning/course-list/`. You can confirm that the browser still shows the same results.

For the *ListView* in the Django official documentation, check this link [Django documentation reference: ListView](#)

get_FOO_display method

get_Foo_display

Model field with the choices option e.g.,

```
:  
LEVEL=[  
    ('B','Basic'),  
    ('I','Intermediate'),  
    ('A','Advanced'),  
]  
:
```

```
{{ course.level }}
```



WEB APPLICATION BASICS

B

HTML & CSS INTRODUCTION

A

```
{{ course.get_level_display }}
```



WEB APPLICATION BASICS

Basic

HTML & CSS INTRODUCTION

Advanced

:

When you use the choices option in your model, the data shown in a browser may be the key to the choices (not the human-readable name). The `get_FOO_display` method can change it to the human-readable name.

FOO is an often-used word as a placeholder for a value that can change. In the Django template context, it represents an object attribute that is defined in the model field with the choices option.

To display a normal model field

`{{ object.FOO }}`



To display a human-readable name for the field
with the choices option

`{{ object.get_FOO_display }}`

Practice

Objective:

Add a human-readable name to the choices option

In the `LearningCourse` model, '`level`' field is defined with the `choices` option. We are adding a human-readable name for the field.

1. Add the object and attribute in a normal way

Add the **yellow** line of the code below into the `course_list.html` file, and save the file.

`templates/employee_learning/course_list.html`

```
:  
<p> {{ course.title|upper }}</p>  
<p> {{ course.level }}</p>  
:
```

Go to `localhost:8000/employee-learning/course-list/`. You can see only B or A, which are keys for the list for the choices option.

LIST Page

WEB APPLICATION BASICS

B

HTML & CSS INTRODUCTION

B

HTML & CSS ADVANCED

A

2. Add the `get_Foo_display` method

Replace the '`level`' part with '`get_level_display`'.

`templates/employee_learning/course_list.html`

```
:  
<p> {{ course.title|upper }}</p>  
<p> {{ course.get_level_display }}</p>  
:
```

Go to localhost:8000/employee-learning/course-list/. You can confirm that the list has changed to human-readable names.

LIST Page

WEB APPLICATION BASICS

Basic

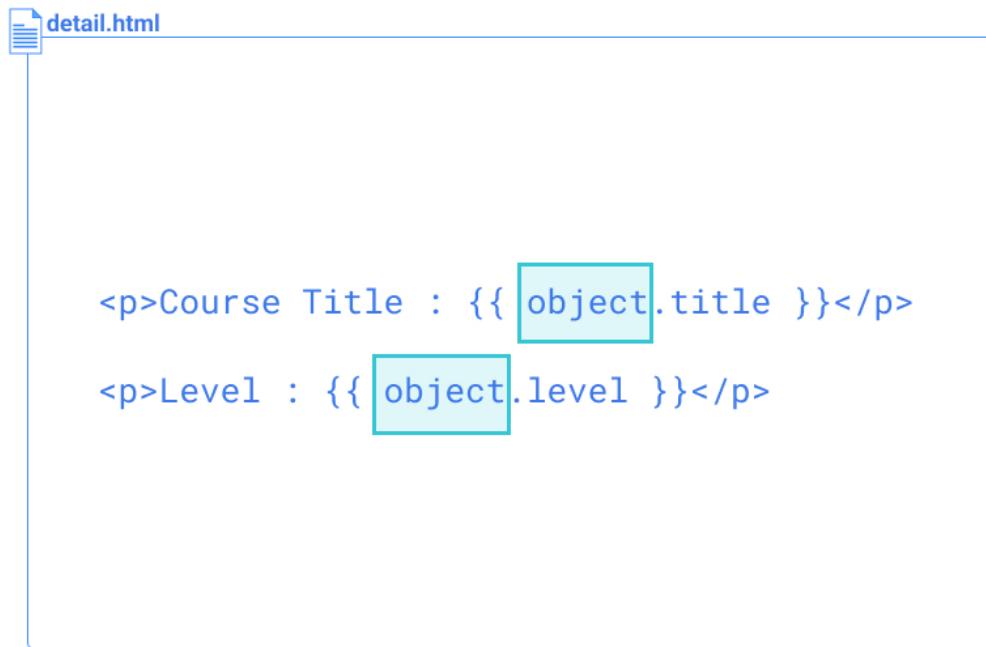
HTML & CSS INTRODUCTION

Basic

HTML & CSS ADVANCED

Advanced

Template for Detail Page



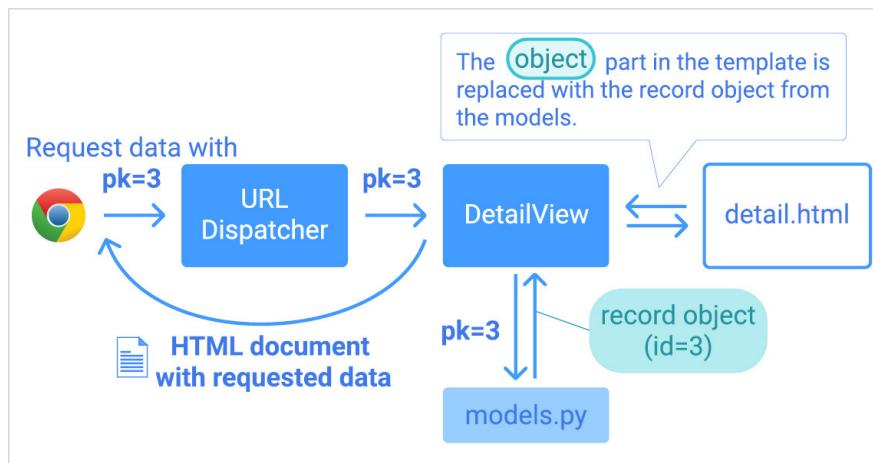
An HTML template for the Detail page is more straightforward. You don't need to use the `for` statement to display multiple objects. The **URL dispatcher** has already processed a complex part of the detail page. On the detail page, you need to show an object for the specified record requested by the URL dispatcher.

How is a detail page displayed?

A brief understanding of the entire process, from an **HTTP request** to an **HTTP response** for a detail page, may help you understand how the detail page template works. The following steps are brief explanations of the processes from an HTTP request to an HTTP response. There are more

detailed steps in between, but this may be enough to roughly understand the detail page template.

1. An HTTP request is generated by a browser. For example, a URL ends with `.../detail/3/`. The last part indicates that the browser is requesting a record object with `id=3`. `pk` as a primary key. The **primary key** is set in the `id` field.
2. **URL Dispatcher** (`urls.py`) dispatches the request to `DetailView`.
3. `DetailView` calls the third record object (`id=3`) through `models.py`.
4. The '`object`' part in `detail.html` is replaced with the record object.
5. `DetailView` returns an HTML document with requested data (record object `id=3`).



Practice

Objective:

Create a template for the detail page

In this practice, we'll create a simple detail page template by implementing the `uppercase` filter and

the `get_FOO_display` method.

1. Edit course_detail.html

Open the `course_detail.html` file, add the **yellow** lines below, and save the file.

`templates/employee_learning/course_detail.html`

```
<h1>DETAIL Page</h1>
<p>Course Title : {{ object.title|upper }} </p>
<p>Level : {{ object.get_level_display }}</p>
```

2. Check the result in a browser

After saving the file, Go to localhost:8000/employee-learning/course-detail/3/. You can change the number at the end. As long as there is data, you'll see a page like the one below.



3. Check the user-friendly name for the object

Like you did for the list page, you can use a user-friendly name for `object`.

In `CourseDetail`, we set `context_object_name = 'course_object'`. Check if it works by updating the code with the **yellow** parts below.

`templates/employee_learning/course_detail.html`

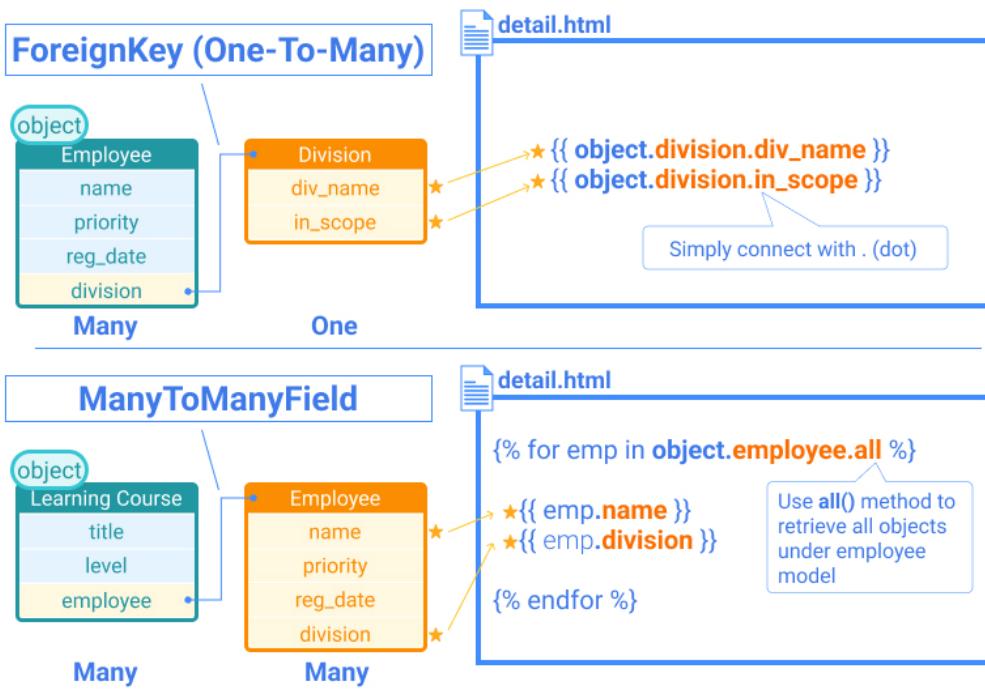
```
:
```

```
<p>Course Title : {{ course_object.title|upper }}</p>
<p>Level : {{ course_object.get_level_display }}</p>
```

Go to localhost:8000/employee-learning/course-detail/3/. You can confirm that the browser still shows the same results.

For the *DetailView* in the Django official documentation, check this link [Django documentation reference: DetailView](#).

Template with Model Relations



When you want to display data from a model with a relationship, you need to slightly adjust how to write a template file.

Modal with ForeignKey

Showing data from a different model connected through `ForeignKey` is relatively straightforward.

In the **ForeignKey (One-To-Many)** case in the main figure, each employee data record has only one division data set. For example, John belongs to the Finance division but not to other divisions.

In this case, you can display data from the related model by connecting field names using . (dot).

For example, when you want to display the `div_name` field through the `Employee` model, you can describe it like `{{ object.division.div_name }}`. In this case, the object represents a record object in the `Employee` model.

Below is an example of handling the template file for the model with `ForeignKey`.

Example:

`employee_detail.html`

```
<h1>ForeignKey Test</h1>

<p>{{ object.name }}</p>
<p>{{ object.priority }}</p>
<p>{{ object.reg_date }}</p>
<p>{{ object.division.div_name }}</p>
<p>{{ object.division.in_scope }}</p>
```

Note: also updated the following files

- `views.py` : added EmployeeDetail view
- `urls.py` : added a URL pattern for the view

In browser

ForeignKey Test

Eleanor Pena
M
March 30, 2023
Product Development
True

Model with ManyToManyField

Showing data from a different model connected through `ManyToManyField` is slightly more complex. As the connected model may have multiple record objects, you need to iterate to deliver all record objects. For example, the Web Application Basic course may have multiple employees assigned.

To display `ManyToMany` Field, you need to do two things

1. Use the `all()` method to call the `employee` object as a `QuerySet`
2. Use the `for` statement

 **detail.html**

```
{% for emp in object.employee.all %}  
    {{ emp.name }}  
    {{ emp.division }}  
{% endfor %}
```

Use `all()` method to
retrieve all objects
under employee
model

Practice

Objective:

Learn how to handle ManyToManyField in the detail page template

In this practice, we'll show different codes for `ManyToManyField` in the detail page template and the corresponding results.

1. Without the `all()` method and the `for` statement

Open the `course_detail.html` file, add the **yellow** lines below, and save the file.

`templates/employee_learning/course_detail.html`

```
:  
<p>Course Title : {{ course_object.title|upper }}</p>  
<p>Level : {{ course_object.get_level_display }}</p>  
  
<p>Assigned Employees : <p>  
<p>{{ object.employee }}</p>
```

Go to one of the detail pages. You'll see that the browser doesn't show the employee data properly.

```
Assigned Employees :  
employee_learning.Employee.None
```

2. Only with the all() method

Change the code to the one below and check the detail page.

templates/employee_learning/course_detail.html

```
:  
<p>Assigned Employees : <p>  
<p>{{ object.employee.all }}</p>
```

Now, you can see that `QuerySet` is displayed, but not in the right format.

```
Assigned Employees :  
<QuerySet [<Employee: Leslie Alexander>, <Employee: Floyd Miles>]>
```

3. Only with the for statement

Change the code to the one below and check the detail page.

templates/employee_learning/course_detail.html

```
:  
<p>Assigned Employees : <p>  
{% for emp in object.employee %}  
<p>{{ emp.name }} : {{ emp.division }}</p>  
{% endfor %}
```

You will encounter an error saying '`ManyRelatedManager` object is not iterable.' This means that Django expects an iterable object like `QuerySet`, but the data set is not iterable.

```
TypeError at /employee-learning/course-detail/1/  
'ManyRelatedManager' object is not iterable
```

4. With both all() method and the for statement

Change the code to the one below and check the detail page.

templates/employee_learning/course_detail.html

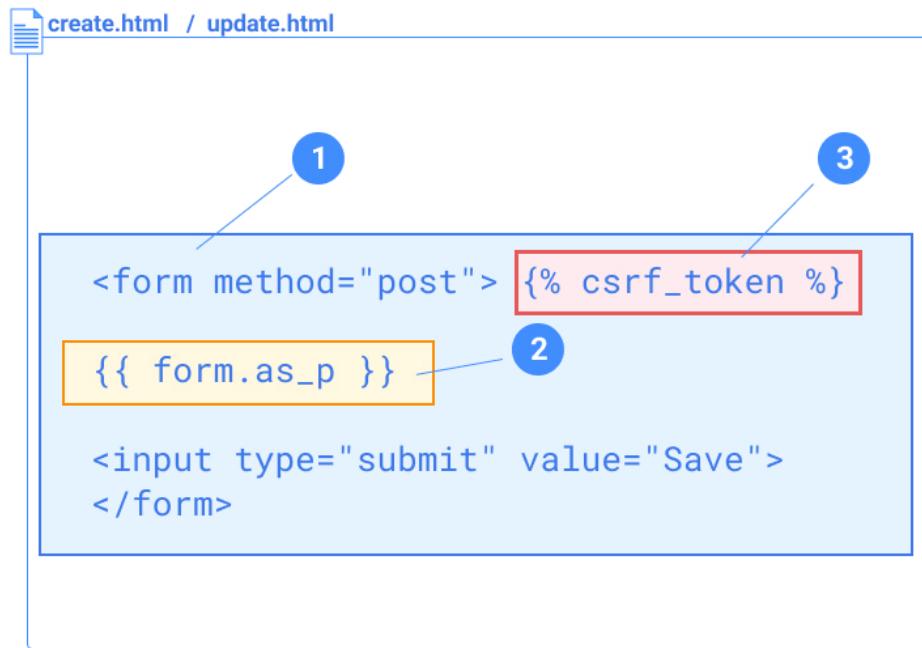
```
:
```

```
<p>Assigned Employees : <p>
{% for emp in object.employee.all %}
<p>{{ emp.name }} : {{ emp.division }}<p>
{% endfor %}
```

Now, you can see that the data are properly displayed.

```
Assigned Employees :
Leslie Alexander : Product Development
Floyd Miles : Product Development
```

Template for Create and Update Page



You can use almost the same templates for the create page and update page. Those two pages should be designed to allow user data input with the form HTML tag.

In the most basic implementation, there are three key parts in the create and update page template.

1. A form with the `<input>` tag
2. `{{ form.as_p }}`
3. `{% csrf_token %}`

1. A form with an input tag

In HTML, the `<form>` tag is used to nest input forms. The `method` attribute defines which HTTP method to be used for input data transfer. For input forms, the **POST method** is usually used.

The `<input>` tag with the `type="submit"` attribute creates a data submission button. You can also use the `<button>` tag instead of the `<input>` tag.

2. {{ form.as_p }}

The `form` part is an object defined by the view. In the `employee_learning` app case, we set the `field` attributes like the one below.

`views.py`

```
fields=('title', 'level', 'employee')
```

Those model fields are converted into input forms through the `form` object.

The `as_p()` method is used to add the `<p>` tags in each form element. There are other similar methods like `as_ul()`, `as_table()`, but `as_p()` that are often used for the form object.

3. {% csrf_token %}

This is used to protect against **Cross-Site Request Forgery (CSRF)**. CSRF is a type of cyberattack that forces an end user to execute unwanted actions on a web application, for example, by sending JavaScript code through the input forms. By adding `{% csrf_token %}`, Django generates a token to build a more secure communication when a user is sending data through the input forms.

If you don't add this tag, when a user submits their input, the user will see the *Forbidden* error message like the one below.

Forbidden (403)

CSRF verification failed. Request aborted.

Help

Reason given for failure:
CSRFTOKEN missing.

In general, this can occur when there is a genuine Cross Site Request Forgery, or when [Django's CSRF mechanism](#) has not been used correctly. For POST forms, you need to ensure:

- Your browser is accepting cookies.
- The view function passes a `request` to the template's `render` method.
- In the template, there is a `{% csrf_token %}` template tag inside each POST form that targets an internal URL.
- If you are not using `CsrfViewMiddleware`, then you must use `csrf_protect` on any views that use the `csrf_token` template tag, as well as those that accept the POST data.
- The form has a valid CSRF token. After logging in in another browser tab or hitting the back button after a login, you may need to reload the page with the form, because the token is rotated after a login.

You're seeing the help section of this page because you have `DEBUG = True` in your Django settings file. Change that to `False`, and only the initial error message will be displayed.

You can customize this page using the `CSRF_FAILURE_VIEW` setting.

Practice

Objective:

Learn how to make create and update page templates

In this practice, we'll make templates for 'create' and 'update' pages.

1. Edit course_create.html and check the result in a browser

Open the `course_create.html` file, add the yellow lines below, and save the file.

`templates/employee_learning/course_create.html`

```
<h1>CREATE Page</h1>
<form method="post"> {{ csrf_token }}
    {{ form.as_p }}
    <input type="submit" value="Save">
</form>
```

After saving the file, Go to `localhost:8000/employee-learning/course-create`.

← → ⌂ ⓘ localhost:8000/employee-learning/course-create/

CREATE Page

Title:

Level: ----- Advanced Intermediate

Employee: Leslie Alexander Bessie Cooper Eleanor Pena Floyd Miles

2. Edit course_update.html and check the result in a browser

Open the `course_update.html` file, add the yellow lines below (the same code as the one for the create page), and save the file.

`templates/employee_learning/course_update.html`

```
<h1>UPDATE Page</h1>
<form method="post"> {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save">
</form>
```

After saving the file, Go to `localhost:8000/employee-learning/course-update/3/`. To go to an update page, you need to specify the `id` number.

← → ⌂ ⓘ localhost:8000/employee-learning/course-update/3/

Update Page

Title:

Level: Advanced Intermediate

Employee: Leslie Alexander Bessie Cooper Eleanor Pena Floyd Miles

Template for Delete Page



```
<form method="post"> {% csrf_token %}

<p>Are you sure you want to delete
"{{ object }}"?</p>

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

</form>
```

The delete page is used to confirm or delete existing data. As the page requires an input form, the template is similar to the create and update page template.

In the most basic implementation, there are three key parts in the delete page template

1. A form with the `<input>` tag
2. `{% csrf_token %}`
3. A confirmation message with the name of the `object` to be deleted

As we already explained about the first two, we'll explain the last one in this section.

`{% object %}`

This part is used to display the name of the object to be deleted. The object name is usually defined in one of the fields in the model set by the following code, which was explained in Chapter 3.

```
def __str__(self):  
    return self.[attribute]
```

Practice

Objective:

Learn how to create a delete page template

In this practice, we'll create a simple delete page template.

1. Edit `course_delete.html` and check the result in a browser

Open the `course_delete.html` file, add the **yellow** lines below, and save the file.

`templates/employee_learning/course_delete.html`

```
<h1> DELETE Page </h1>  
<form method="post">{% csrf_token %}  
    <p>Are you sure you want to delete "{{ object }}"?  
    </p>  
    <input type="submit" value="Confirm">  
</form>
```

After saving the file, Go to localhost:8000/employee-learning/course-delete/3/. To go to the delete page, you need to specify the `id` number.



localhost:8000/employee-learning/course-delete/3/

DELETE Page

Are you sure you want to delete "HTML & CSS Advanced"?

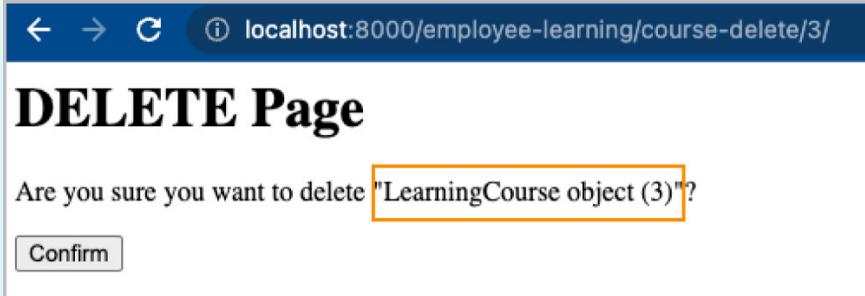
2. Check where the object name comes from

To understand about `{}{{object}}`, update the LearningCourse model in the `>models.py` file. Comment out the **yellow** lines below.

`employee_learning/models.py`

```
class LearningCourse (models.Model):
    :
    # def __str__(self):
    #     return self.title
```

After saving the file, go to the same page in the browser. You can see that the object name has changed to the predefined name with an auto-generated ID.



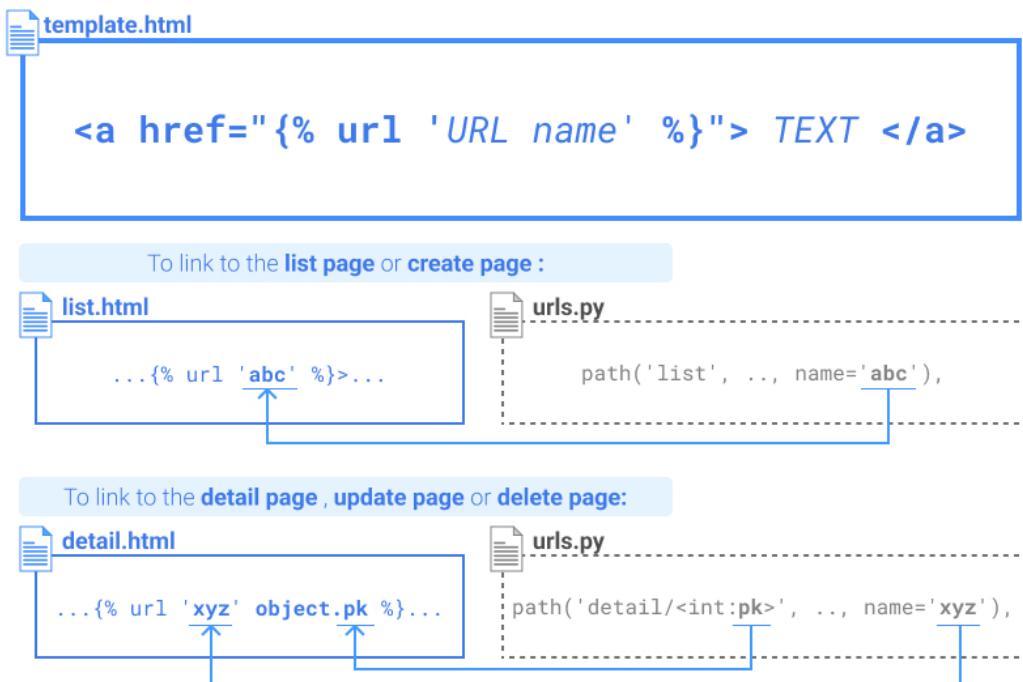
localhost:8000/employee-learning/course-delete/3/

DELETE Page

Are you sure you want to delete "LearningCourse object (3)"?

Remove the comment out hash sign (#) before going to the next section.

Add Links – `{% url %}` tag



You can use `{% url %}` tag to add links to each page. The tag handles **reverse URL resolution** – it specifies URLs by using the **URL name** defined in `urls.py`.

This approach enables you to separate the actual URLs used in browsers from the URL data internally handled in the Django system. When communicating with browsers, `urls.py` converts the internally used URL names into actual URLs.

Basic URL

Creating links to the list page or the create page is simple as they don't have additional variables for specifying an exact page. You can simply use the URL name from the `urls.py` file, as shown in the main figure.

URLs with variables

On the other hand, the detail page, update page, and delete page require more specific information as those pages are linked to each record object. For example, each detail page is defined by the `id`, which is also set as the **primary key** in the database. To create a link to each detail page, you need to pass `pk` information as shown in the main figure.

Practice

Objective:

Learn how to add links in Django templates

We'll add links with the `{% url %}` tags in this practice.

1. Prepare HTML code for links to each page

For efficient operation, prepare a list of links to each page using a text file (any memo app) so that you can copy and paste each code into related document sections.

Use the URL names from the `urls.py` file under the `employee_learning` directory.

Text File

```
link to list page
List

link to detail page
Go Detail

link to create page
Create

link to update page
```

```
<a href="{% url 'course_update' object.pk%}">Update</a>  
link to delete page  
<a href="{% url 'course_delete' object.pk%}">Delete</a>
```

2. Design the link structure

Before pasting the code for links, you need to decide where you want to put them. As the list page and create page are not record-object-specific, you can add the link on any page; however, to make a link to a specific detail, update or delete page, you need to manage how to set a link to them.

Here is the design of the link structure that we apply in this practice.

- **Links to the list page and create page:** put them on all pages, assuming these links will be placed in the navigation bar later.
- **A link to the detail page:** put the link for each data record on the list page so that users can go to the detail page from the list page.
- **Links to the update and delete page:** put them on the detail page so that users can update or delete the data record specific to the detail page.

3. Copy and paste the link code

Based on the link structure design, copy and paste the code.

Links to the list page and create a page in all templates

Paste the code in **yellow** below in the create page template. Do the same for other templates.

templates/employee_learning/course_create.html

```
<a href="{% url 'course_list'%}">List</a>
<a href="{% url 'course_create'%}">Create</a>

<h1> CREATE Page </h1>

<form method="post"> {% csrf_token %}
:
```

Link to the detail page in the list page template

Paste the code in **yellow** below in the list page template. Make sure that you put the code between the for statement. If you paste the code outside of the for statement, the value for pk won't be correctly defined.

templates/employee_learning/course_list.html

```
<h1>LIST Page</h1>

{% for course in object_list %}

<p> {{ course.title|upper }}</p>
<p> {{ course.get_level_display }}</p>

<a href="{% url 'course_detail' course.pk%}">Go
Detail</a>
<hr>
{% endfor %}
```

Links to the update and delete page in the detail page template

Paste the code in **yellow** below in the detail page template.

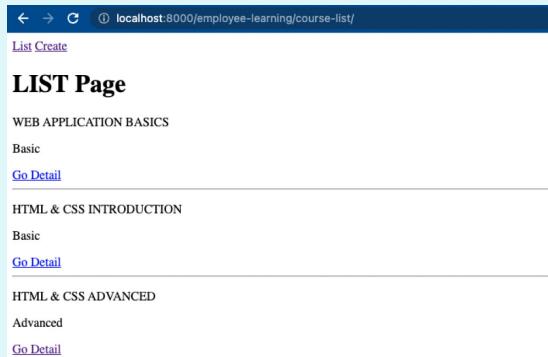
templates/employee_learning/course_detail.html

```
:
<h1> DETAIL Page </h1>
:
{% endfor %}</ul>

<a href="{% url 'course_update'
object.pk%}">Update</a>
<a href="{% url 'course_delete'
object.pk%}">Delete</a>or %}
```

4. Check the results in a browser

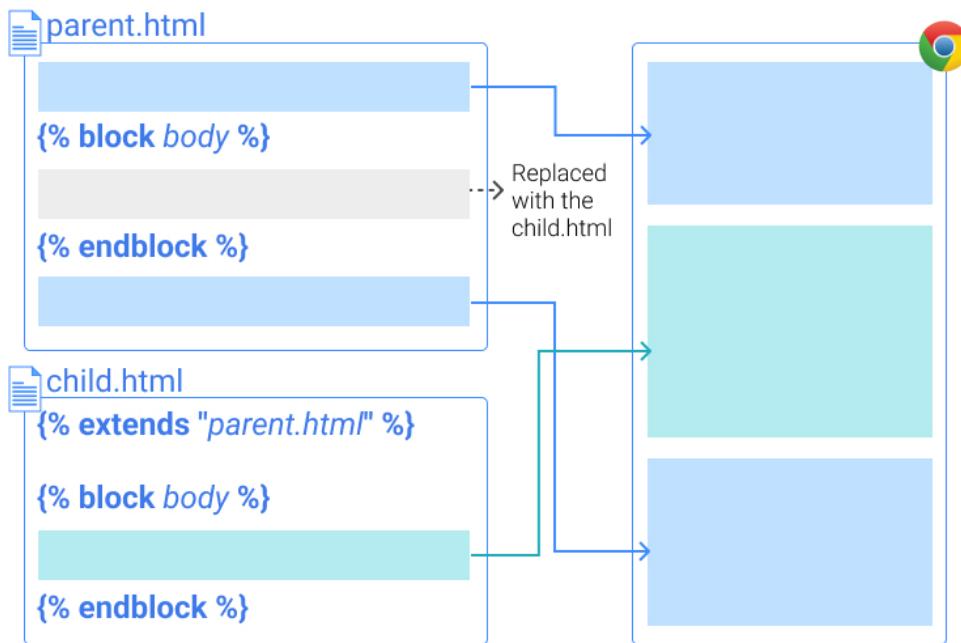
Access to <localhost:8000/employee-learning/course-list/>. You can see that all the links are in place.



The screenshot shows a web browser window with the URL <localhost:8000/employee-learning/course-list/> in the address bar. The page title is "LIST Page". There are three sections of course lists:

- WEB APPLICATION BASICS**
 - Basic
 - [Go Detail](#)
- HTML & CSS INTRODUCTION**
 - Basic
 - [Go Detail](#)
- HTML & CSS ADVANCED**
 - Advanced
 - [Go Detail](#)

Extend Templates – `{% extends %}` tag



In the web page design, there are several pages using a similar design structure. **Django templates** can support the creation of a master template. Then, each page can customize the master template by extending the master template. The `{% extends %}` tag is used to extend the master template.

Here are the key steps to make extended templates.

1. Create `parent.html` (or `base.html`)
2. In the parent file, add code for commonly used sections
3. In the parent file, add `{% block name %}` and `{% endblock %}` for the section that you want to customize
4. Create `child.html`

5. In the child file, add `{% extends "parent.html" %}` in the beginning of the file
6. In the child file, add code between `{% block name %}` and `{% endblock %}`

By implementing this, when the `child.html` template is rendered, `parent.html` and `child.html` are integrated and displayed in the browser as one HTML document.

Practice

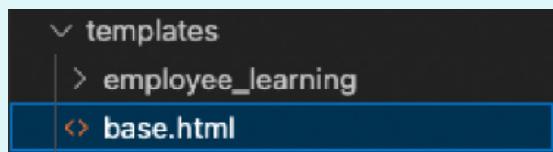
Objective:

Learn how to use template extension

In this practice, we'll demonstrate a template extension.

1. Create `base.html` directly under templates directory

Assuming the base template is used by several applications in the project, we create the `base.html` file directly under the `templates` directory.



2. Edit the `base.html` file

Open the `base.html` file and edit the following three points.

- The first two lines are links to the list page and create page.
- The `{% block body %} ... {% endblock %}` part will be overwritten by the other HTML files.

- The last part is a footer.

templates/base.html

```
<a href="{% url 'course_list'%}">List</a>
<a href="{% url 'course_create'%}">Create</a>

{% block body %}

<p>Insert body contents here</p>

{% endblock %}

<h6>2023 Employee Learning</h6>
```

3. Edit other template files

Open the template file and do the following:

- Add the `{% extends "base.html" %}` tag in the begining
- Delete the links to the list page and the create page that are already written in the `base.html` file.
- Add the `{% block body %}` and `{% endblock %}` tags

The code below is an example of the create page template. The **yellow** lines are the new code.

templates/employee_learning/course_create.html

```
{% extends "base.html" %}

{% block body %}

<h1> CREATE Page </h1>
:
</form>

{% endblock %}
```

4. Check the results in a browser

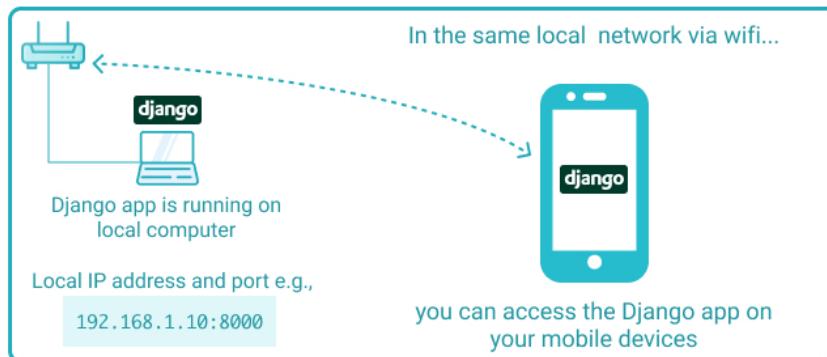
After repeating the same thing for all the files, access to `localhost:8000/employee-learning/course-list/`. You can see that the pages still display the same content, although you have changed the template structure.

Check Developing App UI on Mobile Device

```
python manage.py runserver [Local IP address:Port]
```



```
ALLOWED_HOSTS = ["Local IP address"]
```



During web application development, you may also want to check the in-progress design on mobile devices. Using a local IP address, you can browse the app you are developing. Here are the key steps on how to do it.

1. Check your local IP address

There are several ways to check your local IP address.

Check through the command line

First, you can check it from the command line. Run the `ifconfig` command (`ipconfig` command for Windows). As the command displays a lot of lines, you may need to search it with "192" by

pressing **Command** + **F**. "192" is typically used for local IP addresses.

Command Line - INPUT

```
ifconfig
```

Command Line - RESPONSE

```
lo0: flags=xxx<up,loopback,running,multicast> mtu xxxx
:
inet 192.168.xx.xx netmask 0xffffffff broadcast 192.168.xx.xx
:
```

You can also use the grep command to get the answer directly, like shown below.

Command Line - INPUT

```
ifconfig | grep 192
```

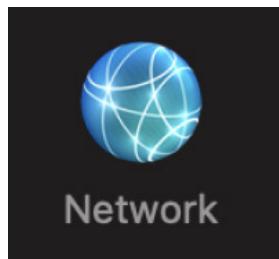
Command Line - RESPONSE

```
inet 192.168.xx.xx netmask 0xffffffff broadcast 192.168.xx.xx
```

Check through System Preferences

You can also check the local IP address in System Preferences on Mac.

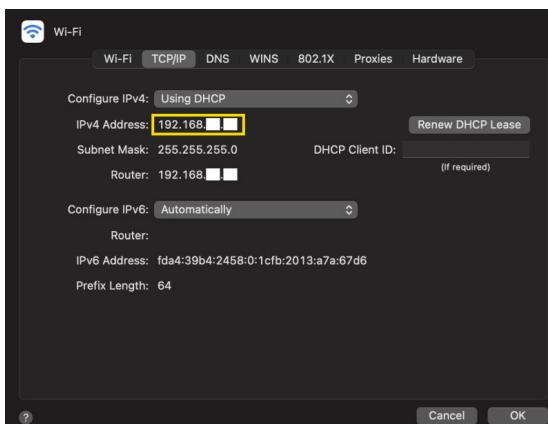
1. Go to System Preferences and click on the Network icon



2. Click on the advanced button on the bottom left

Advanced... ?

3. Go to the wifi setting and check the TCP/IP tab



2. Edit settings.py – add the IP address under ALLOWED_HOSTS

Open the `settings.py` file and edit the `ALLOWED_HOSTS` section like shown below. The reason for also adding "`localhost`" is that you may want to run the `runserver` command using the `localhost` address. If you don't add "`localhost`" while adding another IP address, you cannot use the app using the `localhost` address.

`config/settings.py`

```
ALLOWED_HOSTS = ["localhost", "192.168.xx.xx"]
```

3. Execute the runserver command with specifying IP address and port

Go to the project directory in your terminal and execute the command like shown below.

Command Line - INPUT

```
python manage.py runserver 192.168.86.96:8000
```

4. Check the results in a browser

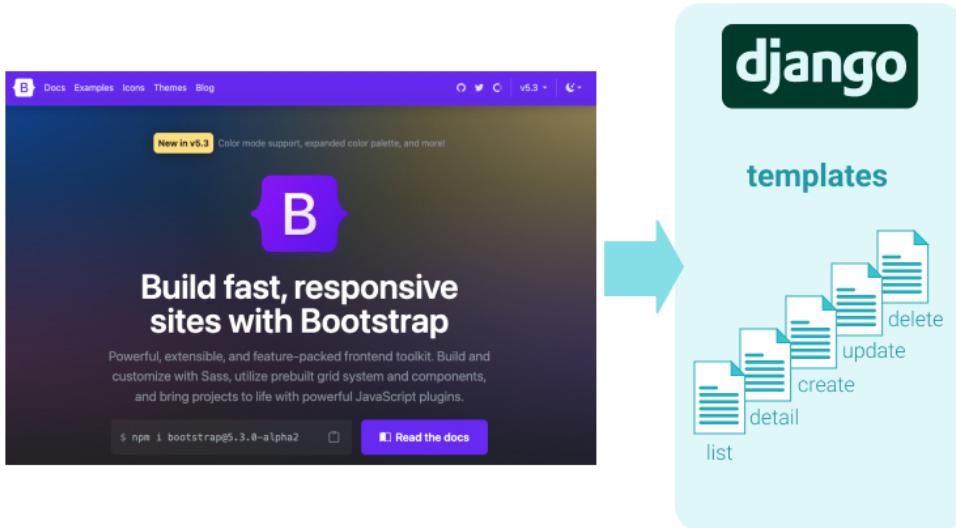
Go to `192.xx.xx.xx:8000/employee-learning/course-list/` in your browser. 192.xxx is your local IP address. You can confirm that the website is accessible on your mobile device.

Note: Make sure your mobile device is connected to the same WiFi as your local computer running the Django application with the `runserver` command.

5. (Optional) Make the local IP address static

Usually, the local IP address is a dynamic IP address defined by **DHCP (Dynamic Host Configuration Protocol)**. Your computer's IP address will be updated regularly. When the IP address changes, you must redo the settings above. If you want to keep the same configuration, make the IP address static in the WiFi setting. However, you need to switch back to DHCP when using other WiFi networks, such as WiFi in a cafe. Otherwise, your laptop may not connect to the new WiFi network.

Django Templates with Bootstrap

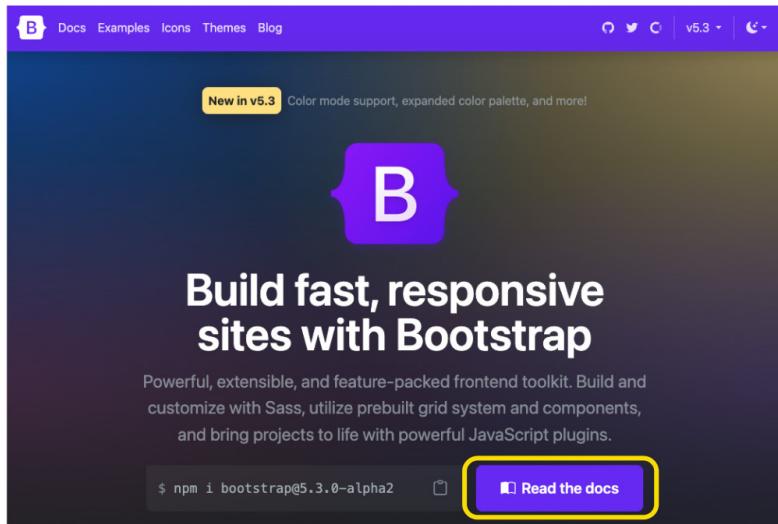


Bootstrap is one of the most popular front-end toolkits. It provides pre-defined CSS and JavaScript code so that you can quickly implement well-designed web pages.

Bootstrap Quick Start

The following are the key steps to implement the basic Bootstrap template.

1. Go to the [bootstrap website](https://getbootstrap.com/) (<https://getbootstrap.com/>) and click on Read the docs button



2. Get code for CDN

Find a CDN version that is easier to implement. Click on the copy button to get the code.

2. Include Bootstrap's CSS and JS. Place the `<link>` tag in the `<head>` for our CSS, and the `<script>` tag for our JavaScript bundle (including Popper for positioning dropdowns, poppers, and tooltips) before the closing `</body>`. Learn more about our [CDN links](#).

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap demo</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha2/dist/c:</head>
  <body>
    <h1>Hello, world!</h1>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha2/dist/</body>
</html>
```

[Copy to clipboard](#)

3. Paste the code into the base.html file and make some adjustments

Paste the code into the `base.html` file and adjust the code like shown below. The yellow lines are from Bootstrap. Keep the body contents from the original `base.html` file and update the website title to *Employee Learning*.

`templates/base.html`

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
    scale=1">
    <title>Employee Learning</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
    alpha2/....>
</head>
<body>

    <a href="#">List</a>
    <a href="#">Create</a>

    {% block body %}
    <p>Insert body contents here</p>
    {% endblock %}

    <h6>2023 Employee Learning</h6>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
    alpha2/....></script>
</body>
</html>

```

4. Check the results

Go to the list page. You can see that the style has been slightly adjusted already.

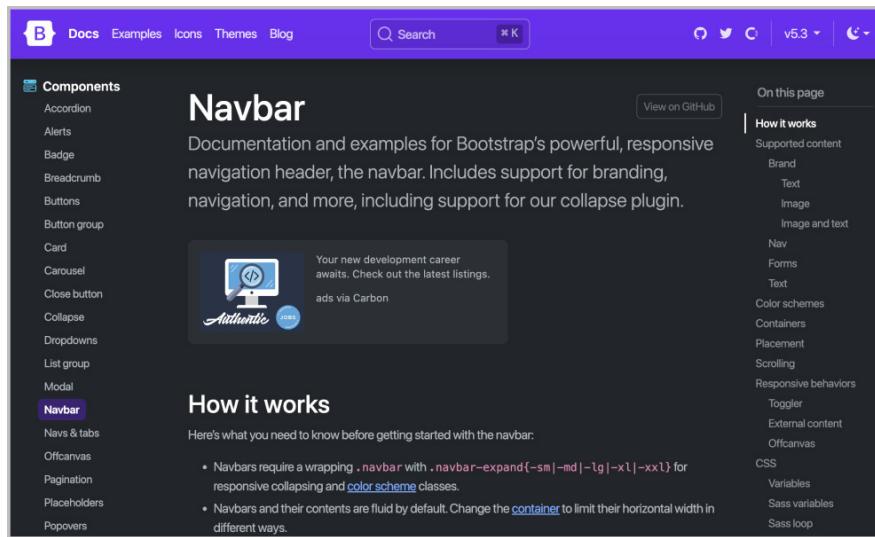
LIST Page	
<u>List</u> <u>Create</u>	
WEB APPLICATION BASICS	
Basic	<u>Go Detail</u>
<hr/>	
HTML & CSS INTRODUCTION	
Basic	<u>Go Detail</u>
<hr/>	
HTML & CSS ADVANCED	
Advanced	<u>Go Detail</u>

Add Navbar (Navigation Bar)

Next, add a navigation bar style to the links on the List page and Create page using the Navbar component from Bootstrap.

1. Go to the Navbar section under Components

Check the left sidebar and find the **Components** section. Under the **Components** section, you can find the **Navbar** component.

A screenshot of the Bootstrap documentation website. The top navigation bar has links for Docs, Examples, Icons, Themes, and Blog. A search bar and a user profile icon are also present. The main content area has a purple header with the word "Components". Below it, a sidebar lists various components: Accordion, Alerts, Badge, Breadcrumb, Buttons, Button group, Card, Carousel, Close button, Collapse, Dropdowns, List group, Modal, and Navbar. The "Navbar" item is highlighted with a purple background. The main content area shows the "Navbar" documentation page. It features a title "Navbar", a "View on GitHub" button, and a "How it works" section with a sub-section "Supported content". There's also a sidebar titled "On this page" containing links to other Bootstrap components like Nav, Forms, Text, Color schemes, Containers, Placement, Scrolling, Responsive behaviors, Toggler, External content, Offcanvas, CSS, Variables, Sass variables, and Sass loop.

2. Copy the code for one of the Navbar components

There are several sets of code with different sub-components (e.g., dropdown or search bar). You can use the relevant ones for your app. This section will use a simple design without a dropdown or search bar.

```

Navbar Home Features Pricing Disabled
Copy to clipboard
HTML

<nav class="navbar navbar-expand-lg bg-body-tertiary">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item">
          <a class="nav-link active" aria-current="page" href="#">Home</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Features</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Pricing</a>
        </li>
        <li class="nav-item">
          <a class="nav-link disabled">Disabled</a>
        </li>
      </ul>
    </div>
  </div>
</nav>

```

3. Paste the code into the base.html file and make some adjustments

Paste the code into the *base.html* file (after the `<body>` tag). Adjust the code like shown below. The **white** code is from the original code of *base.html*, and the **yellow** code is from Bootstrap. Replace the `href` and the text part with the ones from the original *base.html* file. Also, add the brand name of this web app "Training Registration".

`templates/base.html`

```

<body>

  <nav class="navbar navbar-expand-lg bg-body-tertiary">
    <div class="container-fluid">
      <a class="navbar-brand" href="#">Training Registration</a>
      <button class="navbar-toggler" type="button" ...>
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarNav">

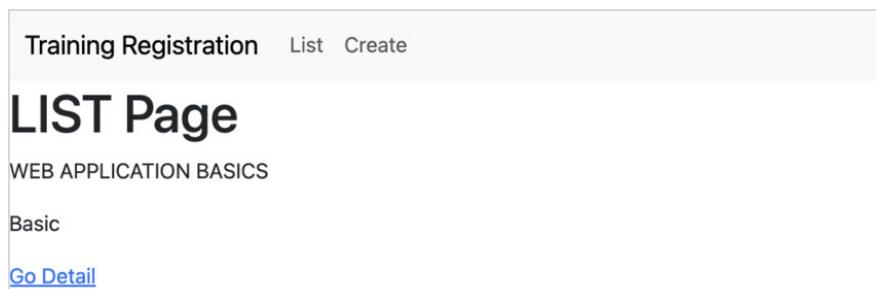
```

```
<ul class="navbar-nav">
  <li class="nav-item">
    <a class="nav-link" href="{% url
      'course_list'%}">List</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="{% url
      'course_create'%}">Create</a>
  </li>
</ul>
</div>
</div>
</nav>

{% block body %}
```

4. Check the results

Go to the list page. You can see that the navigation bar style has already been adjusted.



Also, shrink the browser window size horizontally. As Bootstrap applies the **responsive design** concept, you can see that the **navbar** menu items are collapsed under the hamburger menu.

Training Registration

List

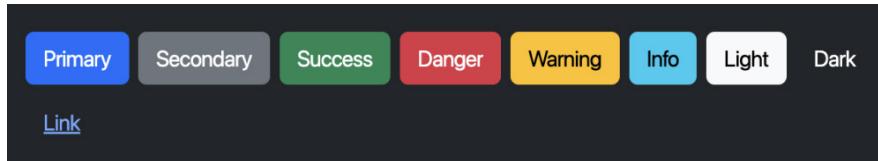
Create

LIST Page

WEB APPLICATION BASICS

Style buttons

Bootstrap also provides quick styling code for buttons. You can choose a color of a button by the color theme name such as primary or secondary.



1. Find code for Bootstrap Buttons

Code for buttons is available under the **Button** section under **Components** on the left sidebar.

The screenshot shows the Bootstrap documentation page for 'Components'. The left sidebar has a 'Buttons' category selected. The main content area is titled 'Variants' and discusses button variants. It shows a row of colored buttons labeled 'Primary', 'Secondary', 'Success', 'Danger', 'Warning', 'Info', 'Light', and 'Dark'. Below this is a 'Link' button. A code block shows the HTML for these buttons:

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>

<button type="button" class="btn btn-link">Link</button>
```

2. Pick a color theme for each button

Before adjusting code, it's better to come up with a design principle first. The following is the guidance for color theme selections.

List page

- "**Go Detail**" button: `primary`

Detail page

- "**Update**" button: `warning`
- "**Delete**" button: `danger`

Create page

- "**Save**" button: `primary`

Delete page

- "**Confirm**" button: `danger`

3. Copy the class code and paste into the tag to style

At this time, you don't need to copy the entire code. Just copy only the class. For example, `class="btn btn-primary"` for the primary color buttons.

Copy the class code and paste it into the relevant tags. For example, to adjust the Save button on the Create page, add the class with the **yellow** line below.

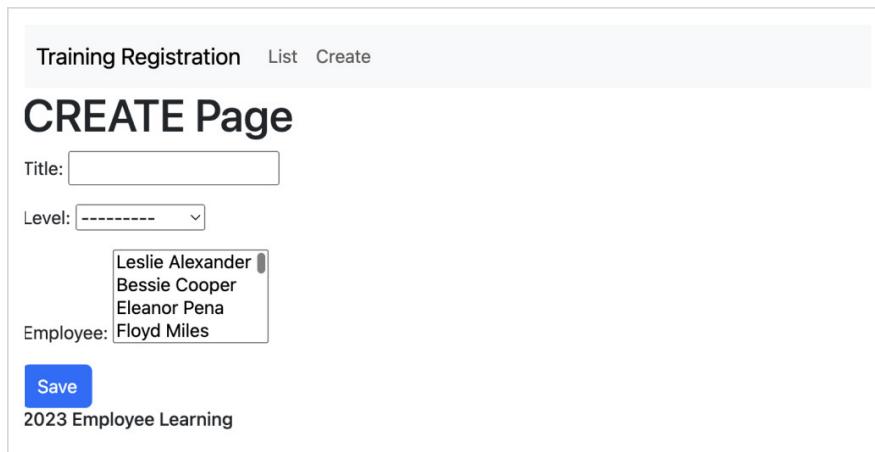
`templates/course_create.html`

```
<input type="submit" value="Save" class="btn btn-primary">
```

Do the same for all buttons based on the design principle.

4. Check the results

Go to the Create page. You can see that the button is already nicely styled.



Style the List page with a Card and Flex box

On the list page, there are multiple objects. Using Bootstrap, you can quickly make a card-style layout.

1. Find code for Bootstrap Cards

Code for Cards is available in the Cards section under Components on the left sidebar.

The screenshot shows the Bootstrap documentation website. The top navigation bar includes links for Docs, Examples, Icons, Themes, and Blog. A search bar and social media icons for GitHub and Twitter are also present. On the left, a sidebar titled 'Components' lists various components: Accordion, Alerts, Badge, Breadcrumb, Buttons, Button group, Card (which is highlighted in purple), Carousel, and Close button. The main content area features a large heading 'Cards' and a sub-headline: 'Bootstrap's cards provide a flexible and extensible content container with multiple variants and options.' Below this is a card component example with a placeholder image and text: 'Your new development career awaits. Check out the latest listings.' and 'ads via Carbon'.

Select one of the card styles and copy the code. In this practice, choose the basic one but delete the image like shown below.

The screenshot shows a code editor interface. On the left, there is a preview of a card component with a gray image placeholder labeled 'Image cap', a 'Card title', and some sample text. Below the preview is a blue button labeled 'Go somewhere'. On the right, there is an 'HTML' tab containing the corresponding HTML code. A small 'Copied!' message with a clipboard icon is visible near the bottom right of the code area. The code is as follows:

```
<div class="card" style="width: 18rem;">
  
  <div class="card-body">
    <h5 class="card-title">Card title</h5>
    <p class="card-text">Some quick example text to build on the card title and make up the bulk of the card's content.</p>
    <a href="#" class="btn btn-primary">Go somewhere</a>
  </div>
</div>
```

2. Paste the code into the course_list.html file and make some adjustments

Paste the code into the list.html file (after the `{% for %}` tag). Adjust the code like shown below.

`templates/course_list.html`

```
{% for course in object_list %}

<div class="card" style="width: 18rem;">
```

```

<div class="card-body">
    <h5 class="card-title">{{ course.title|upper }}</h5>
    <p class="card-text">{{ course.get_level_display }}</p>
    <a href="{% url 'course_detail' course.pk%}" class="btn
        btn-primary">Go Detail</a>
</div>
</div>

{% endfor %}

```

The white code is from the original code of *base.html*, and the **yellow** code is from Bootstrap. Basically, adjust the title and text. As we have already styled the button, keep the code for the button as it is.

At this stage, the UI is not really well-design yet.

The screenshot shows a web application interface. At the top, there's a navigation bar with the text "Training Registration" and links for "List" and "Create". Below this is a section titled "LIST Page". This section contains two items, each represented by a card-like box. The first item is labeled "WEB APPLICATION BASICS" and has the status "Basic". The second item is labeled "HTML & CSS INTRODUCTION" and also has the status "Basic". Each item has a blue rectangular button labeled "Go Detail" at the bottom.

3. Adjust the layout and sizes

Usually, adjusting the layout and sizes requires meticulous work. **Bootstrap utilities** are helpful tools for speeding up layout and size adjustments using short class names.

The code below is an example of layout and size adjustments. The **yellow** lines add class or style attributes.

`templates/course_list.html`

```

{% block body %}

<div class="container text-center">
    <h1 class="text-center m-5">LIST Page</h1>

```

```

[{"for course in object_list"}]
<div class="card m-1 d-inline-flex" style="width: 18rem;
height:10rem;">
<div class="card-body">
<h5 class="card-title h-50">{{ course.title|upper }}</h5>
<p class="card-text">{{ course.get_level_display }}</p>
<a href="{% url 'course_detail' course.pk%}" class="btn btn-primary p-0 d-block w-50 mx-auto" style="height:1.2rem; font-size:0.8rem">Go Detail</a>
</div>
</div>
[{"endfor"}]
</div>

[{"endblock"}]

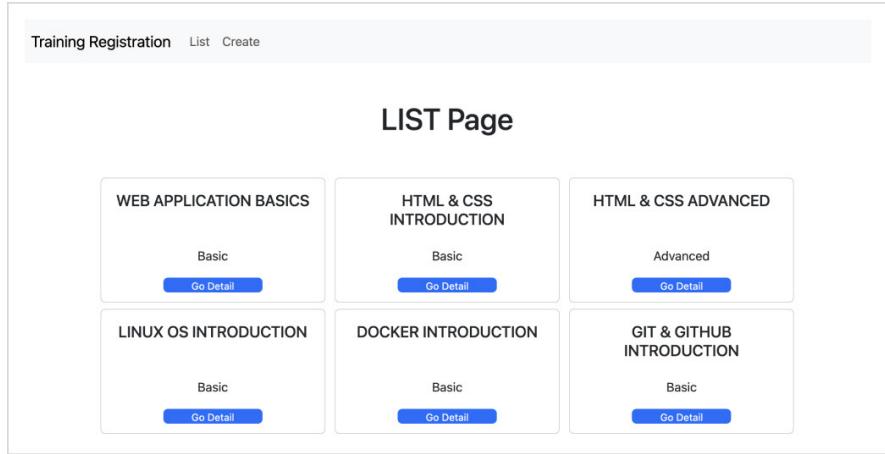
```

Here are some explanations of Bootstrap utilities.

- **text-center**: Text utility to move content or nested elements to the center horizontally
- **m-#**: Spacing utility to adjust margin. You can specify numbers between 0 and 5.
- **d-inline-flex**: Display or Flex utility to change the element to flex inline box.
- **h-#**: Sizing utility to adjust the height of the element relative to the parent element. 50 means 50% of the parent element's height.
- **p-#**: Spacing utility to adjust padding. You can specify numbers between 0 and 5. 0 means zero padding.
- **d-block**: Display utility to change the element to a block element.
- **w-#**: Sizing utility to adjust the width of the element relative to the parent element. 50 means 50% of the parent element's width.
- **mx-auto**: Spacing utility to horizontally center fixed-width block-level content

We are also using the **container** class to wrap all elements, which sets a **max-width** at each responsive breakpoint and allows appropriate margins on the right and left edges.

Now, you can see that the layout and sizes are nicely set.



Style the Detail and Delete page

In this part, we will style the detail page using Bootstrap utilities. The **yellow** parts in the code below are the updated parts. The text and spacing utilities help you quickly adjust positions and sizes.

`templates/course_detail.html`

```
{% block body %}

<div class="container text-center">

    <h1 class="text-center m-5"> DETAIL Page </h1>
    <p class="text-muted mb-0">Course Title</p>
    <h5> {{ object.title|upper }} </h5>
    <p class="text-muted mt-3 mb-0">Level</p>
    <h5>{{ object.get_level_display }}</h5>
    <p class="text-muted mt-3 mb-0">Assigned Employees</p>
    {% for emp in object.employee.all %}
        <p class="mb-0">{{ emp.name }} : {{ emp.division }}</p>
    {% endfor %}

    <a href="{% url 'course_update' object.pk%}" class="btn btn-warning m-3">Update</a>
    <a href="{% url 'course_delete' object.pk%}" class="btn btn-danger m-3">Delete</a>

</div>

{% endblock %}
```

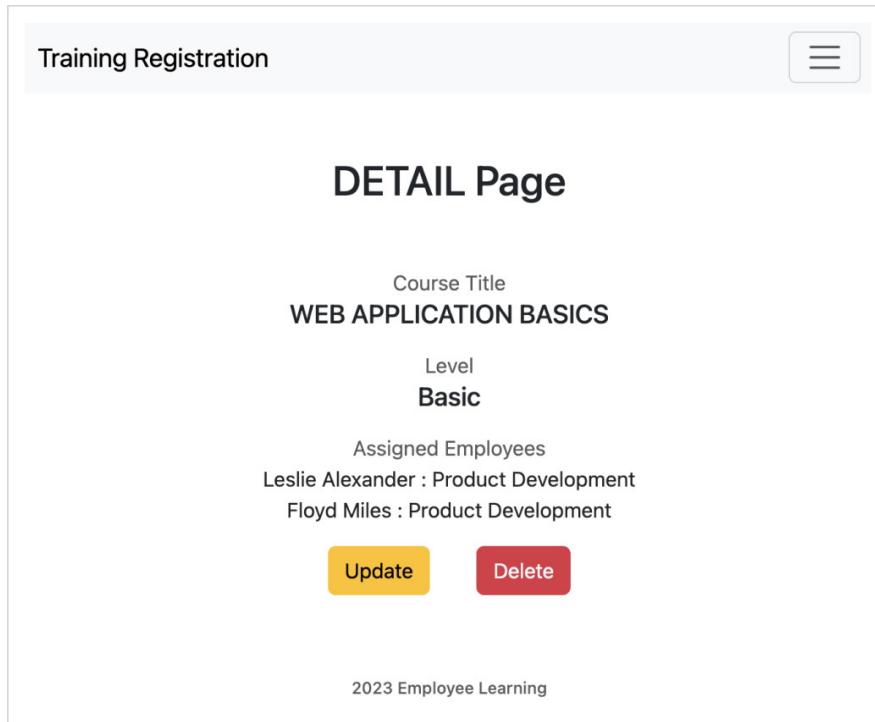
Also, update the footer in the `base.html` file with the **yellow** code below.

templates/base.html

```
{% endblock %}

<h6 class="text-muted text-center mt-5" style="font-size:0.8rem">2023 Employee Learning</h6>
```

Now, you can see a better Detail page design.



Next, update the Delete page with the **yellow** code below.

templates/course_delete.html

```
{% block body %}

<div class="container text-center">

    <h1 class="text-center m-5"> DELETE Page </h1>

    <form method="post">{% csrf_token %}
        <p>Are you sure you want to delete "{{ object }}"?</p>
        <input type="submit" value="Confirm" class="btn btn-danger">
    </form>

</div>
{% endblock %}
```



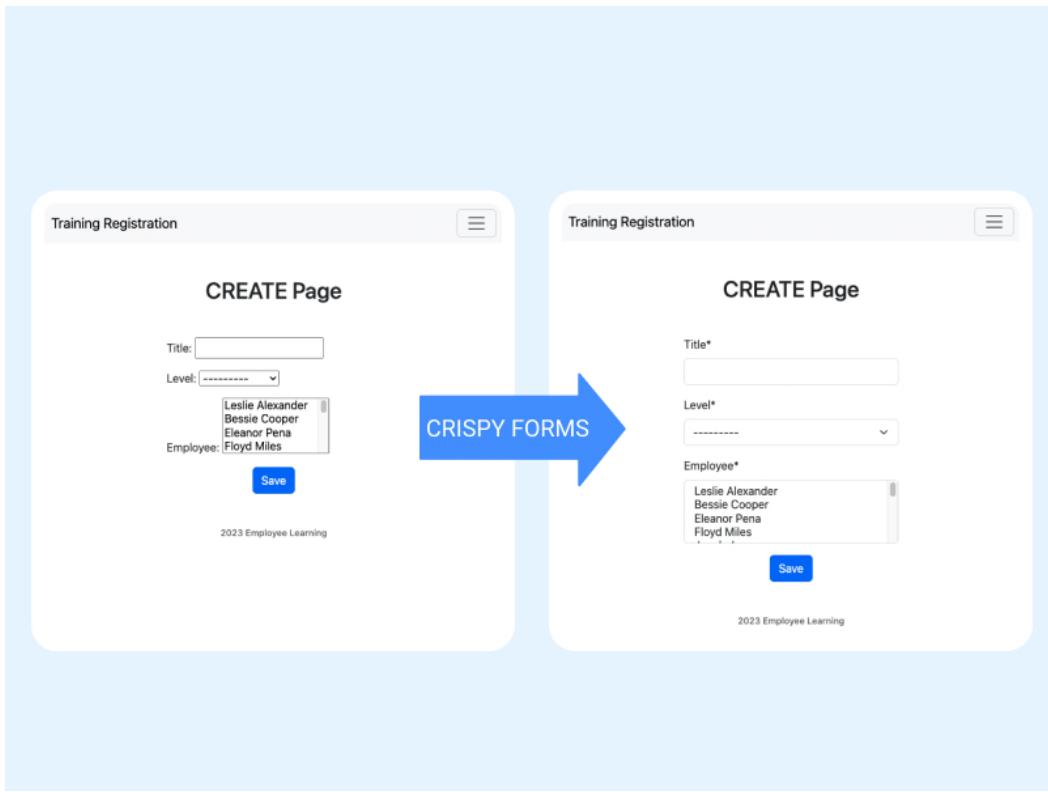
DELETE Page

Are you sure you want to delete "HTML & CSS Introduction"?

Confirm

2023 Employee Learning

Crispy Forms



As Bootstrap cannot handle Django auto-generated input forms directly, styling the create and update pages requires an additional tool called **Crispy Forms**. Here are the key steps to implement Crispy Forms.

1. Install Crispy Forms and Bootstrap 5 Template pack

First, add two library names in the requirements.txt file.

- *django-crispy-forms*
- *crispy-bootstrap5*

requirements.txt

```
Django==4.1.7
django-crispy-forms
crispy-bootstrap5
```

Run the `pip` command to install the two libraries.

Command Line - INPUT

```
(d_env) project_d % | pip install -r requirements.txt
```

2. Update settings.py

Update the `settings.py` file like shown below.

config/settings.py

```
# template directory path
TEMPLATE_DIR = BASE_DIR / 'templates'

CRISPY_ALLOWED_TEMPLATE_PACKS = "bootstrap5"
CRISPY_TEMPLATE_PACK = "bootstrap5"

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'test_app',
    'employee_learning',
    'crispy_forms',
    'crispy_bootstrap5',
]
```

3. Update the Create and Update page templates

To use Crispy Forms, you need two adjustments.

- Add the `{% load crispy_forms_tags %}` tag to load Crispy Forms
- Change `{{ form.as_p }}` to `{{ form|crispy }}`

Also, you can add some classes and styles to adjust the layout and sizes. For the create and update forms, we added `min-width` and `max-width` to prevent the forms from expanding or shrinking too much horizontally – try to make sizing nice and responsive.

The code below is an example of the changes in the Create page template. You can do the same for the Update page template.

`templates/employee_learning/course_create.html`

```
{% extends "base.html" %}  
{% load crispy_forms_tags %}  
{% block body %}  
  
<div class="container w-50" style="min-width:300px; max-width:  
500px;">  
    <h1 class="m-5 text-center"> CREATE Page </h1>  
    <form method="post"> {% csrf_token %}  
        {{ form|crispy }}  
        <input type="submit" value="Save" class="btn btn-primary  
d-block mx-auto">  
    </form>  
</div>  
  
{% endblock %}
```

4. Check the results

To check the results in your browser, run the `runserver` command again.

Command Line - INPUT

```
(d_env) project_d % | python manage.py runserver
```

You can see that the Create and Update pages are nicely designed now.



CREATE Page

Title*

Level*

Employee*

- Leslie Alexander
- Bessie Cooper
- Eleanor Pena
- Floyd Miles

Save

2023 Employee Learning

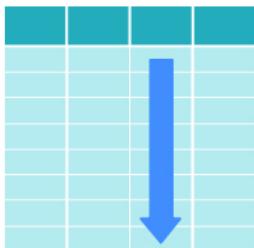
[Library documentation reference: Django Crispy Forms](#)

Customize Views (1) – Change List Order

views.py

```
queryset = ModelName.objects.order_by('field')
```

Change data order



Normal order

```
.order_by('field')
```

e.g., .order_by('title')

Reverse order

```
.order_by('-field')
```

e.g., .order_by('-title')

Using the `queryset` attribute and the `order_by` method, you can easily change the order of the list. Here, we'll explain how to use them with the employee learning app example.

Change the list data order

To change the order of the list on the List page based on the course title, edit the `CourseList` view by adding the **yellow** part of the code below. Also, comment out the `model` part as `queryset` and `model` have similar functionality.

`employee_learning/views.py`

```
class CourseList(ListView):
    # model = LearningCourse
    queryset = LearningCourse.objects.order_by('title')>
    :
```

You can see that the order of the list is changed based on the course title.

AAAA	BBBB	DOCKER ADVANCED & KUBERNETES	DOCKER INTRODUCTION
Intermediate Go Detail	Basic Go Detail	Advanced Go Detail	Basic Go Detail
GIT & GITHUB INTRODUCTION Basic Go Detail	HTML & CSS ADVANCED Advanced Go Detail	HTML & CSS INTRODUCTION Basic Go Detail	JAVASCRIPT INTRODUCTION Basic Go Detail
LINUX OS INTRODUCTION Basic Go Detail	PYTHON BASICS Basic Go Detail	WEB APPLICATION BASICS Basic Go Detail	ZZZZ Basic Go Detail

Reverse order

To reverse the order, you need to just add '-' before the field name.

`employee_learning/views.py`

```
class CourseList(ListView):
    # model = LearningCourse
    queryset = LearningCourse.objects.order_by('-title')
    :
```

You can see that the order of the list is reversed.

LIST Page

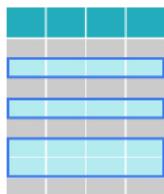
ZZZZ	WEB APPLICATION BASICS	PYTHON BASICS	LINUX OS INTRODUCTION
Basic Go Detail	Basic Go Detail	Basic Go Detail	Basic Go Detail
JAVASCRIPT INTRODUCTION	HTML & CSS INTRODUCTION	HTML & CSS ADVANCED	GIT & GITHUB INTRODUCTION
Basic Go Detail	Basic Go Detail	Advanced Go Detail	Basic Go Detail
DOCKER INTRODUCTION	DOCKER ADVANCED & KUBERNETES	BBBB	AAAA
Basic Go Detail	Advanced Go Detail	Basic Go Detail	Intermediate Go Detail

Customizing Views (2) – Filter Lists



```
queryset =  
    ModelName.objects.filter(field='keyword')
```

Filter data



contains 'keyword'	➡ filter(field__contains='keyword')
start with 'keyword'	➡ filter(field__startswith='keyword')
end with 'keyword'	➡ filter(field__endswith='keyword')
greater than 'number or string'	➡ filter(field__gt='keyword')
greater than & equal to 'number or string'	➡ filter(field__gte='keyword')
lower than 'number or string'	➡ filter(field__lt='keyword')
lower than & equal to 'number or string'	➡ filter(field__lte='keyword')

Using the `queryset` attribute and the `filter` method, you can filter the list shown on the list page. Here we'll explain how to use them with the employee learning app example.

Filter objects with the field that matches a keyword

To display only basic-level courses, edit the `CourseList` view by adding the **yellow** part of the code below. Also, comment out the previous one.

`employee_learning/views.py`

```
class CourseList(ListView):  
    # model = LearningCourse  
    # queryset = LearningCourse.objects.order_by('-title')  
    queryset = LearningCourse.objects.filter(level='B')  
    :
```

You can see that only basic courses are shown.

LIST Page			
WEB APPLICATION BASICS Basic Go Detail	HTML & CSS INTRODUCTION Basic Go Detail	LINUX OS INTRODUCTION Basic Go Detail	DOCKER INTRODUCTION Basic Go Detail
GIT & GITHUB INTRODUCTION Basic Go Detail	JAVASCRIPT INTRODUCTION Basic Go Detail	PYTHON BASICS Basic Go Detail	BBBB Basic Go Detail

Customize filtering

The filter method also has the functionality to customize filtering approaches.

Here are some examples:

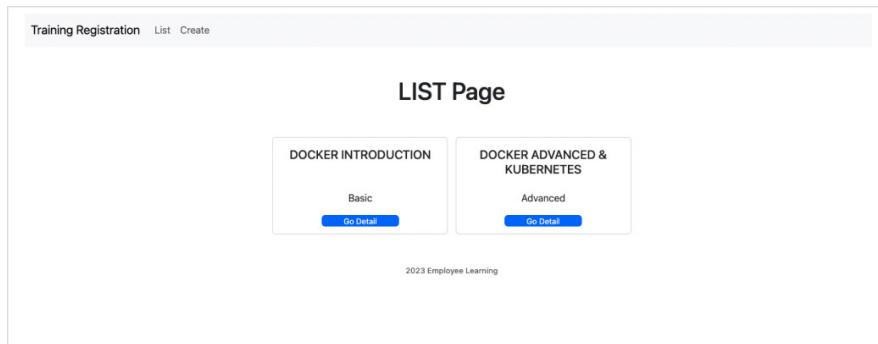
- `filter(field__contains='keyword')`: Filter objects that contain 'keyword'
- `filter(field__startswith='keyword')`: Filter objects that start with 'keyword'
- `filter(field__endswith='keyword')`: Filter objects that end with 'keyword'
- `filter(field__gt='number or string')`: Filter objects that are greater than 'number or string'
- `filter(field__gte='number or string')`: Filter objects that are greater than or equal to 'number or string'
- `filter(field__lt='number or string')`: Filter objects that are lower than 'number or string'
- `filter(field__lte='number or string')`: Filter objects that are lower than or equal to 'number or string'

For example, if you want to get the courses whose names include '*Docker*', use `filter(title__contains='Docker')`. To implement this, edit the CourseList view by replacing the **yellow** part of the code below.

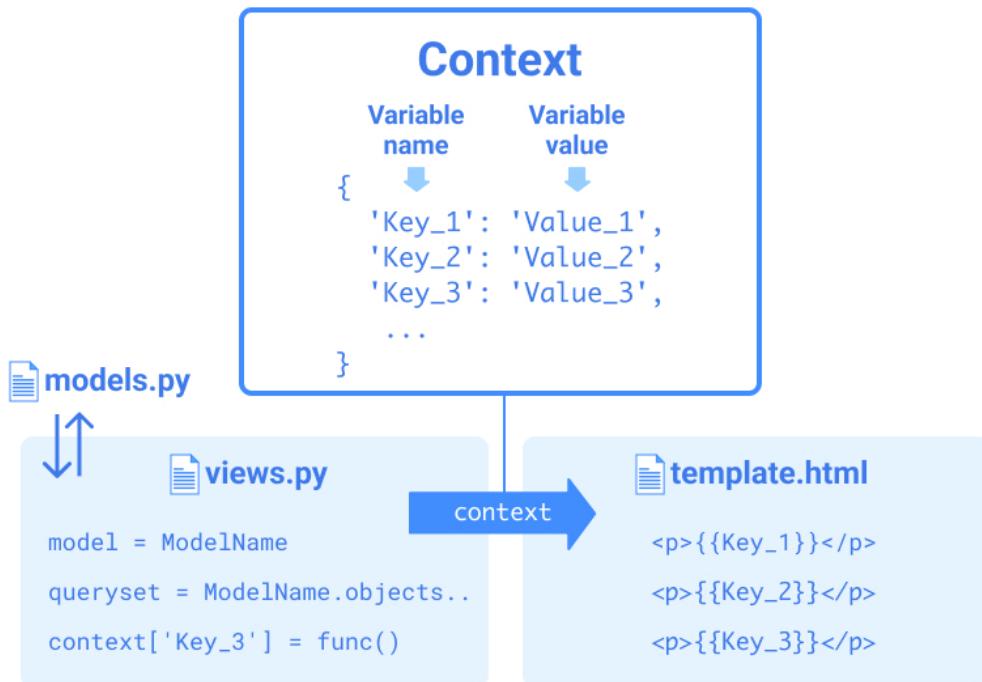
`employee_learning/views.py`

```
class CourseList(ListView):
    # model = LearningCourse
    # queryset = LearningCourse.objects.order_by('-title')
    queryset =
        LearningCourse.objects.filter(title__contains='Docker')
    :
```

You can see that only *Docker* courses are shown.



Context



Context is a frequently used word when you write views. It is a set of dictionary-type data mapping variable names to variable values that are transferred from views to templates.

Check context data in your terminal

To understand the concept of context, let's print the context data for the list page.

Add the **yellow** part in the CourseList view code, which redirects the output (the context data for the list page) to your terminal.

`employee_learning/views.py`

```
class CourseList(ListView):
    # model = LearningCourse
    # queryset = LearningCourse.objects.order_by('-title')
    queryset =
        LearningCourse.objects.filter(title__contains='Docker')
```

```
:  
def get_context_data(self, **kwargs):  
    context = super().get_context_data(**kwargs)  
    print(context)
```

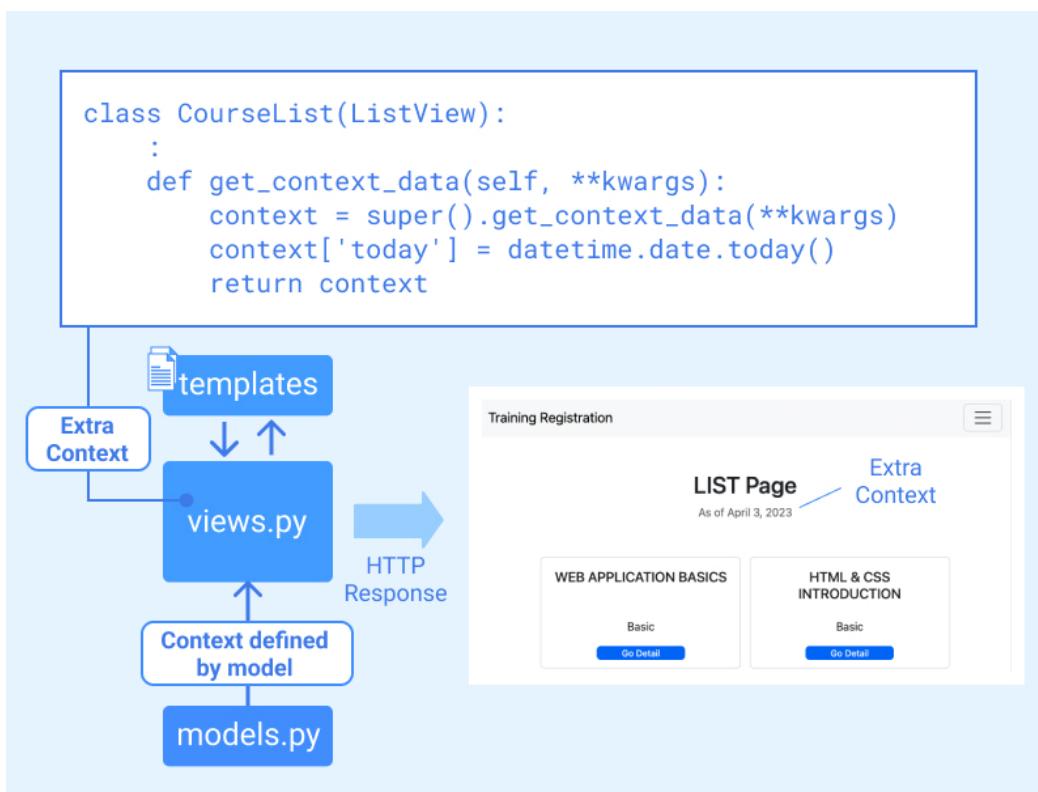
You don't see any list items when you visit the list page in your browser. Instead, you'll get output in your terminal like shown below. (In the actual case, there are no line breaks. We added line breaks to make the code easy to read.)

Command Line - RESPONSE

```
[04/Apr/2023 01:43:32] "GET /employee-learning/course-list/  
HTTP/1.1" 200 1748  
{'paginator': None,  
'page_obj': None,  
'is_paginated': False,  
'object_list': <QuerySet [<LearningCourse: Docker Introduction>,  
<LearningCourse: Docker Advanced & Kubernetes>]>,  
'course_object_list': <QuerySet [<LearningCourse: Docker  
Introduction>, <LearningCourse: Docker Advanced & Kubernetes>]>,  
'view': <employee_learning.views.CourseList object at  
0x1106026d0>}
```

You can see that the context carries several data sets that are used to display list items on the list page.

Customize Views (3) – Add Extra Context



The `context` data are already generated when you specify the `model` or `queryset` attribute. By overriding the `get_context_data` method, you can also add new context data. Here we'll explain how to add extra context with the employee learning app example.

1. Override the `get_context_data` method

In this case, we'll add new context data '`today`'. In `views.py`, add import the `datetime` function so that you can use it in the overriding method.

In the `CourseList` view, override the `get_context_data` method like shown in the `yellow` code part below.

`employee_learning/views.py`

```
from django.urls import reverse_lazy
import datetime

class CourseList(ListView):
    :
    queryset =
        LearningCourse.objects.filter(title__contains='Docker')
    :
def get_context_data(self, **kwargs):
    context = super().get_context_data(**kwargs)
    context['today'] = datetime.date.today()
    return context
```

2. Add the new context in the template file

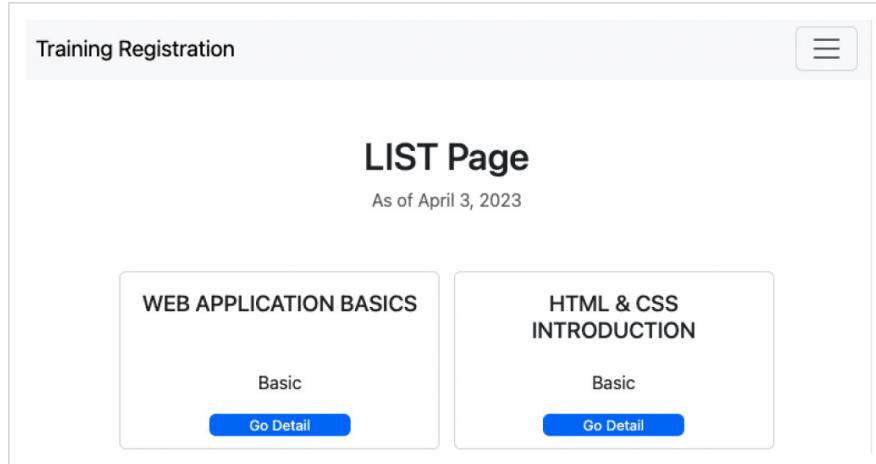
Now, a new context, '`today`', is available. To display it on the List page, update the `course_list.html` file. Add the **yellow** part of the code below.

`templates/employee_learning/course_list.html`

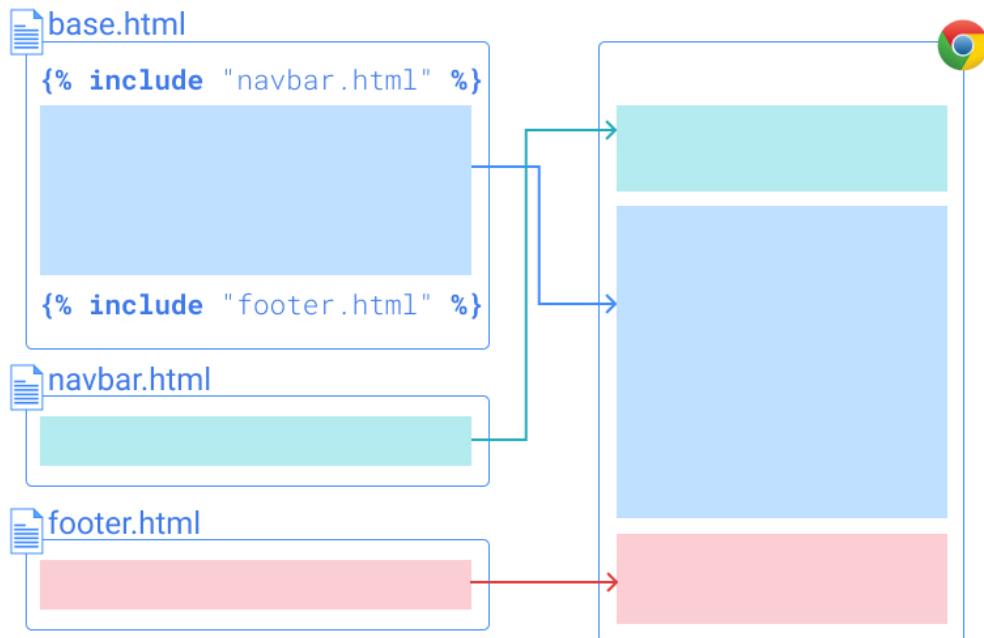
```
{% block body %}

<h1 class="text-center mt-5">LIST Page</h1>
<p class="text-muted text-center mb-5">As of {{ today }}</p>
:
```

When you go to the list page, you can see that the extra context data is displayed on the list page.



Modularize Templates – `{% include %}` tag



In web page design, several design components can be re-utilized on several pages, such as the **navigation top bar** or **footer**. Django templates can support the elimination of redundancy by modularizing templates. The `{% include %}` tag is used to include the modular parts in a template file.

Practice

Objective:

Learn how to use modular Django templates

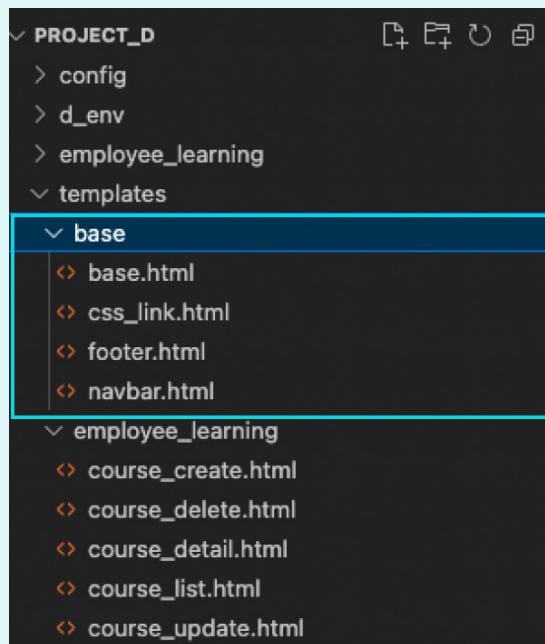
In this practice, we'll demonstrate how to modularize template files.

1. Create module templates

We'll create three module templates.

- *css_link.html*
- *navbar.html*
- *footer.html*

For better folder management, create a base directory under the templates directory and place the *base.html* and the three new HTML files. The directory structure will be like the one below.



2. Adjust other template files to keep the link with *base.html*

As you moved the *base.html* file, you need to adjust the path to the file in other template files. Edit each of the following files.

- *course_create.html*
- *course_delete.html*
- *course_detail.html*
- *course_list.html*
- *course_update.html*

Add the **yellow** part of the code to the top of each file.

templates/employee_learning/course_list.html

```
:
{%
block body %}
{% extends "base/base.html" %}
:
```

3. Move the code to each module template from **base.html**

Cut out each part of the code and paste it into each template file like shown below.

templates/base/css_link.html

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0
-alpha2/....>
```

templates/base/navbar.html

```
<nav class="navbar navbar-expand-lg bg-body-
tertiary">
    <div class="container-fluid">
        :
    </div>
</nav>
```

templates/base/footer.html

```
<h6 class="text-muted text-center mt-5"
style="font-size:0.8rem">2023 Employee
Learning</h6>
```

```
<script  
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-  
alpha2/...."></script>
```

4. Add {%- include %} tags in base.html

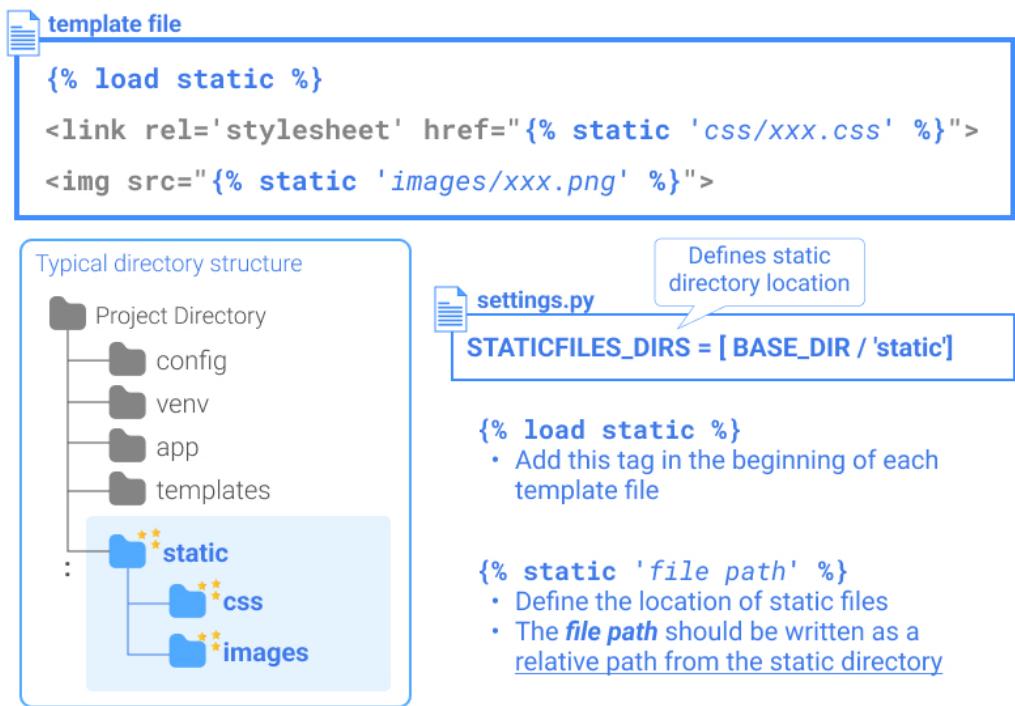
Add {%- include %} tags with related HTML paths in the place where you cut the code. The *base.html* file becomes very simple, as shown below.

templates/base/base.html

```
<!doctype html>  
<html lang="en">  
<head>  
    <meta charset="utf-8">  
    <meta name="viewport" content="width=device-  
    width, initial-scale=1">  
    <title>Employee Learning</title>  
  
    {% include "base/css_link.html" %}  
  
</head>  
  
<body>  
    {% include "base/navbar.html" %}  
  
    {% block body %}  
        <p>Insert body contents here</p>  
    {% endblock %}  
  
    {% include "base/footer.html" %}  
  
</body>  
</html>
```

Check the browser. As the actual code doesn't change, each page should stay the same.

Static Files in Development Environment – `{% static %}` tag



Static files are files that don't change dynamically, such as image, video, CSS, and JavaScript files. These files are typically saved in a particular location for efficient resource handling.

In the production environment, handling static files is slightly complicated. This lesson will cover static file handling in the development environment.

Static directory location

In Django, the `static` directory is usually placed directly under the project directory with sub-directories such as `css`, `images` or `js`. The `static` directory path should be registered in the `settings.py` file so that Django can handle static files properly. The example below

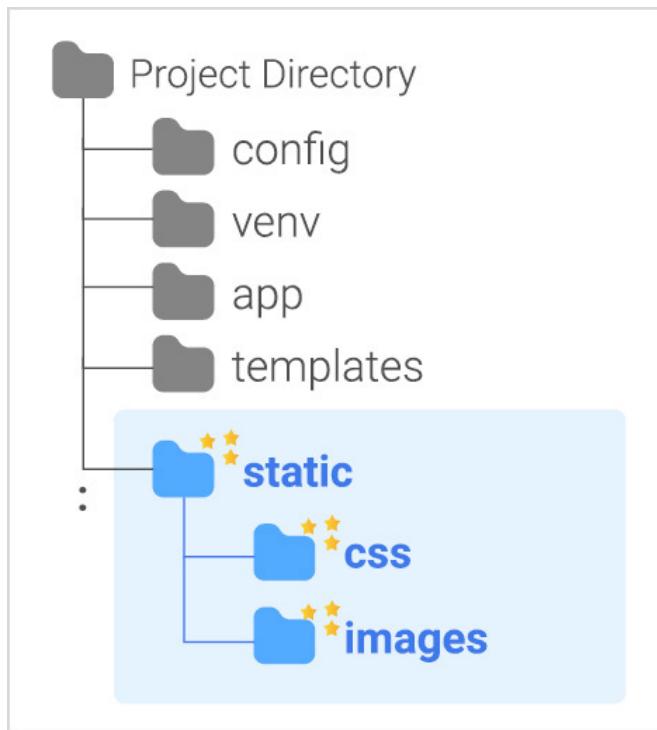
shows how you need to add the `static` directory path to `settings.py`.

config/settings.py

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

Create a static directory tree

The frequently used static files are image files and CSS files. Add the directory tree like shown below.



{% static %} tag

In Django template files, `{% static %}` tags are used to access the static files stored under the static directory. There are two rules

- Add `{% load static %}` tag at the beginning of each template file. With this tag, the template file loads static files. In extended template files, you also need to put this tag even though the parent file has this tag.

- Use `{% static 'file path' %}` tag where you need to define the static file path. The file path should be written as a **relative path from the static directory**

Practice 1

Objective:

Learn how to add a CSS file

In this practice, we'll demonstrate how to add a custom CSS file.

1. Create static directories and edit settings.py

Follow the instructions for creating required directories and editing `settings.py` in the main section above.

2. Create a CSS file

Create the `custom.css` file under the css directory, and add a simple code like the one below.

`static/css/custom.css`

```
body {  
    background-color: lightblue  
}
```

3. Edit css_link.html

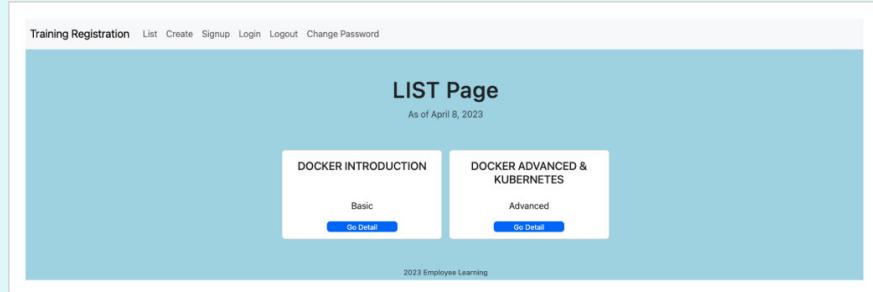
Add the `<link>` tag to make a link with the new `custom.css` file.

`templates/base/css_link.html`

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

4. Check the result

You can see that the background color has changed to light blue.



As this is for testing purposes, change the color back to white and save the CSS file.

Practice 2

Objective:

Learn how to add images to Django templates

In this practice, we'll demonstrate how to add icons on the create, update, and delete page.

1. Prepare image files

In this practice, we'll use the three icons below. You can find similar ones on the free icon sites. Save the icons under the *images* directory under the static directory.



2. Edit HTML files

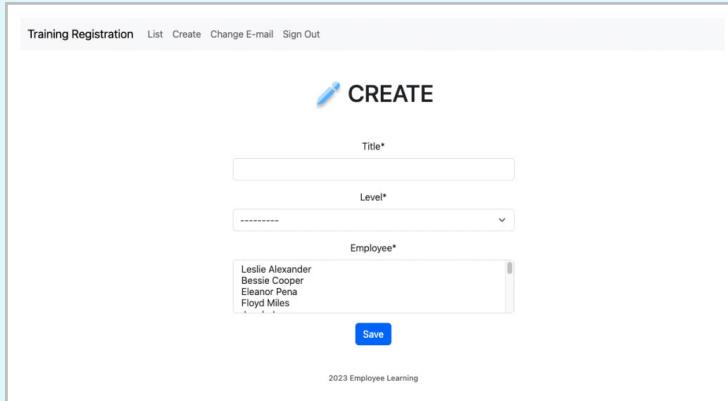
Add the yellow part of the code below in the `course_create.html` file. `{% load static %}` should be after `{% extends "base/base.html" %}`. Also, change the title name from "`CREATE Page`" to "`CREATE`" considering the title size balance.

`templates/employee_learning/course_create.html`

```
{% extends "base/base.html" %}  
{% load static %}  
  
{% load crispy_forms_tags %}  
  
{% block body %}  
  
<h1 class="m-5 text-center"> CREATE </h1>  
  
:
```

3. Check the result

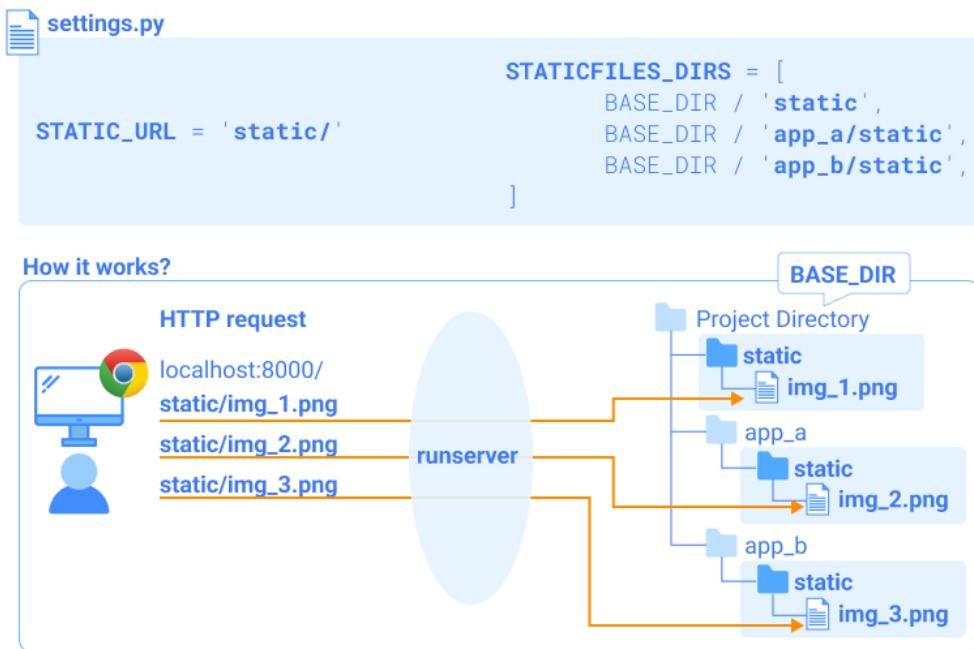
Save the file and go to the Create page. You can see that the icon was successfully added.



The screenshot shows a web page titled "CREATE" with a pencil icon. It has three input fields: "Title*", "Level*", and "Employee*". The "Employee*" dropdown menu lists four names: Leslie Alexander, Michael Cooper, Eleanor Pena, and Floyd Miles. A "Save" button is at the bottom.

Do the same for the update and delete pages.

STATIC_URL and STATICFILES_DIRS



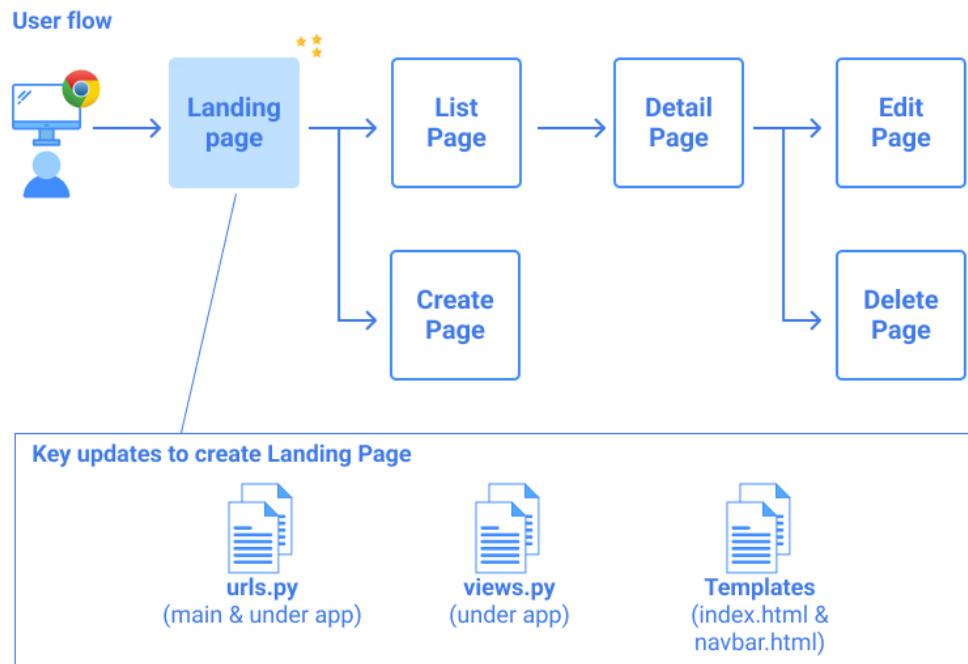
`STATIC_URL` defines the URL that is linked with the project directory location(s) defined by `STATICFILES_DIRS`. We'll explain how `STATIC_URL` and `STATICFILES_DIRS` work in this lesson.

The `static` directory is usually set directly under the project directory, but you can also set it under each app. You can set multiple `static` directories, and Django handles the `static` files in different `static` directories as if they are under the same directory.

For example, when there is an HTTP request '`localhost:8000/static/img-1.jpg`' like in the main figure, Django gets the resource from the `static` directory under the *project directory*.

When there is an HTTP request '`localhost:8000/static/img-2.jpg`', Django gets the resource from the `static` directory under the `app_a` directory.

Create Index HTML



In the previous section, we created **CRUD** pages but haven't made a home page yet. Usually, *index.html* is used for the home page.

To create the home page, you need to update the *urls.py* files to redesign the URL structure, add a view for the home page and update template files.

Redesign URL structure (Update the urls.py files)

This exercise may be a good for understanding how to design **URL patterns** in the *url.py* files. In the current structure, the main *url.py* under the *config* directory is the first touch point when an HTTP response comes. Then, it includes the *urlpatterns* written in *url.py* under the *employee_learning* app.

The main urls.py

First, you need to adjust the way `employee_learning` app's paths can be included. We used '`employee-learning`' as a part of the URL to make a clear contrast between `test_app` and `employee_learning` for an explanation purposes, but you can leave that part blank so that you can define the view of the index page in `urls.py` in the `employee_learning` app.

After the steps above are completed, URLs of the pages for the `employee_learning` app become shorter as you don't need to use full paths of the directory tree.

To execute the change, update the file like in the example below.

config/urls.py

```
:  
urlpatterns = [  
    path('', include('employee_learning.urls')),  
    path('admin/', admin.site.urls),  
    path('test-app/', include('test_app.urls')),  
    # path('employee-learning/',  
    include('employee_learning.urls')),  
]
```

The app urls.py

Next, you need to define the path and view of the home page. As the home page should be accessible when a user accesses the domain itself, there is no need to add an additional path. For the view, use `Index`, which will be defined in the next step.

employee_learning/urls.py

```
from .views import ..., Index  
:  
urlpatterns = [  
    path('', Index.as_view(), name='index'),  
    path('course-list/', CourseList.as_view(),  
        name='course_list'),  
    :  
]
```

Create Index view

As each dynamic page should have a view, we need to make a view even for a simple home page.

Here is an example code. To use *TemplateView*, import it and define the template file (*index.html*). And add one extra context to show a date.

employee_learning/views.py

```
from django.views.generic import ListView, DetailView,
CreateView, UpdateView, DeleteView, TemplateView
:
class Index(TemplateView):
    template_name = 'index.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['today'] = datetime.date.today()
        return context
```

Update Template files

Create the index.html file

First, you need to create a new HTML file for the home page. Create *index.html* under the templates directory and edit the file like shown below.

templates/index.html

```
{% extends "base/base.html" %}

{% block body %}



# Digital Training Registration



As of {{ today }}

View Registered Programs
Create a New Program



{% endblock %}
```

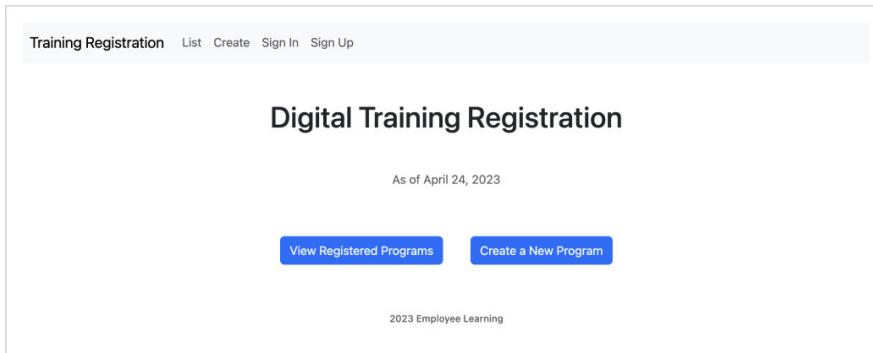
Edit the navbar.html

To make a link to the home page on the brand logo in the navigation bar, edit the navbar.html file.

templates/base/navbar.html

```
<a class="navbar-brand" href="{% url 'index'%}">Training  
Registration</a>
```

Go to *http://localhost:8000/*. Now, you won't see the yellow page anymore, and you can go to the home page instead.



Chapter 5

User Management

This chapter covers the basics of **user authentication** and how to develop user management functions using **Django Allauth**.

The following topics are covered in this chapter.

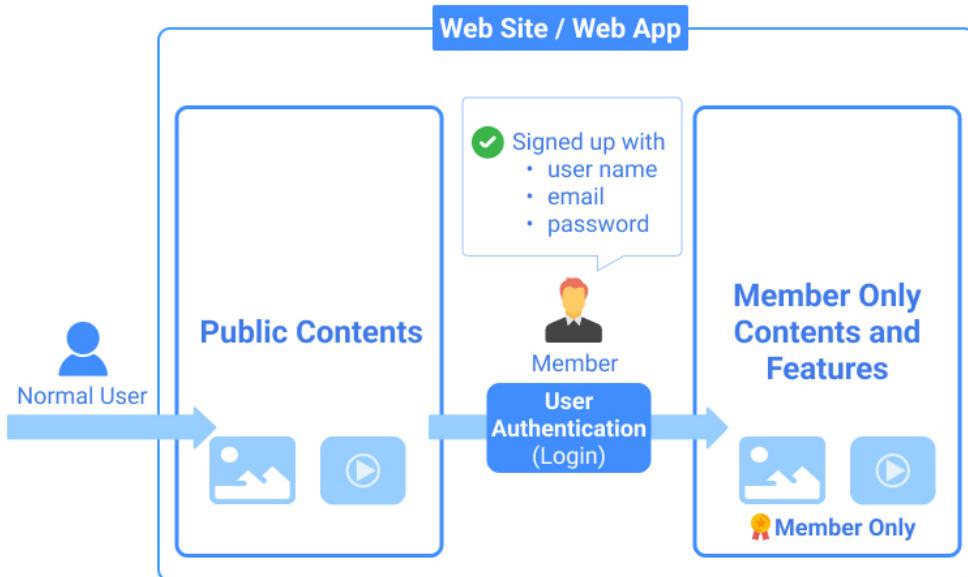
Topics

1. User Authentication
2. Overview of User Management Functions
3. User Management Function Development with Django
4. Approaches to Building User Management Functions in Django
5. Django Allauth (1) – Introduction
6. Django Allauth (2) – Installation and Initial Settings
7. Django Allauth (3) – Email Verification via Console
8. Django Allauth (4) – Email Verification via Gmail

- [**9. Django Allauth \(5\) – Social Login with GitHub**](#)
- [**10. Django Allauth \(6\) – Social Login with Google**](#)
- [**11. Django Allauth \(7\) – Allauth Template File Setup**](#)
- [**12. Django Allauth \(8\) – Add Basic Styling with Bootstrap and Crispy Forms**](#)
- [**13. Django Allauth \(9\) – Customize Sign-in and Sign-up Pages**](#)
- [**14. User Models**](#)
- [**15. Login Required – LoginRequiredMixin**](#)
- [**16. User Login Status Icon on Navigation Bar**](#)

User Authentication

User Authentication



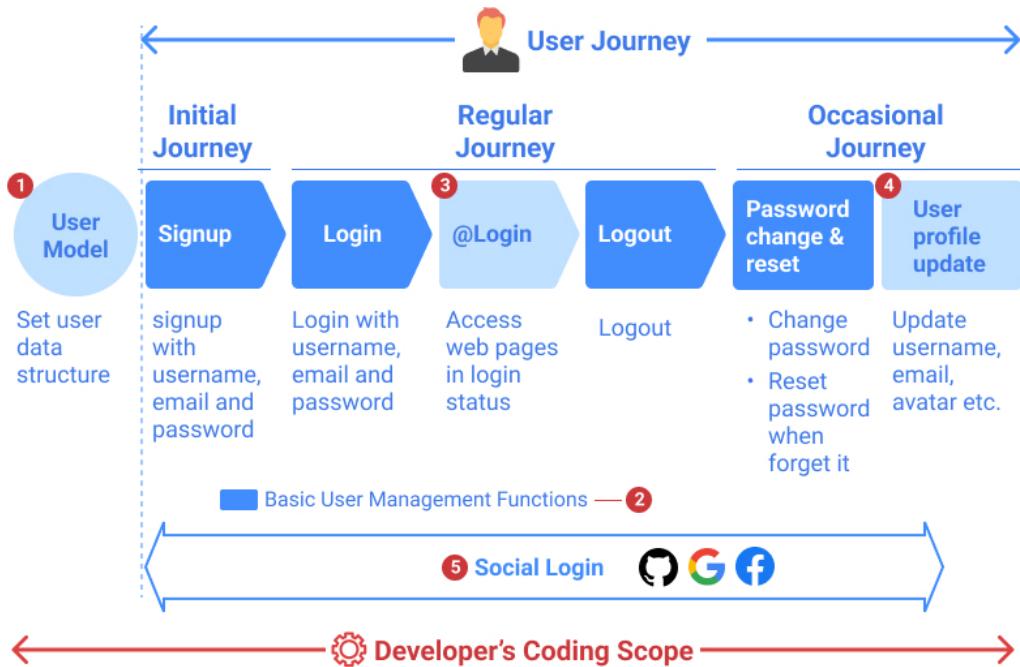
Many web applications usually require user signup. The objectives of having users sign up vary from providing member-only content to customizing services based on the users' data.

When a user signs up for a web application service, the web application registers the user's essential information, including user name, email address, and password (user identity).

When the user logs in to the service, the web application checks if the user is the same person who has previously signed up for the service. The process of verifying the identity of the user is called **user authentication**.

Overview of User Management Functions

Overview of User Management Functions



To build the user management functions, you need to understand the overall picture of the user management functions. To have a comprehensive understanding, it is good to observe the end-to-end process from two perspectives:

1. User journey perspective
2. Developer's perspective

User Journey Perspective

From the user journey perspective, there are three types of journeys.

- **Initial journey:** start to use the app by signing up for the service
- **Regular journey:** the primary cycle of the user journey – login, a login user journey, and logout.
- **Occasional journey:** when required, a user can change their password, or when they forget the password, they can reset their password. Also, the user can update their user profile, name, and email address.

Developer's Perspective

From the developer's perspective, you need to design and build the user management functions to support the user journey. There are five key design points in user management functions from the developer's perspective.

1. User Model

First, you need to design what user data the app will handle. Django provides a built-in user model including the following fields:

- `username`
- `first_name`
- `last_name`
- `email`
- `password`
- `groups`
- `user_permissions`
- `is_staff`
- `is_active`
- `is_superuser`

- `last_login`
- `date_joined`

You can also add more fields, such as a user icon and other user profiles, by creating a custom user model.

2. Basic User Management Functions

When we talk about the user management functions for web applications, the following five essential components are often discussed.

- **Signup:** register user data in the user model
- **Login:** authenticate the user to allow them to use the service
- **Logout:** exit the service
- **Password change:** change the user password because of reasons other than the user losing the password (such as security reasons)
- **Password reset:** reset the password when the user forgets their password

In web applications, each function is handled on a different page, but the designs of these pages are pretty standardized.

3. Login Status

Once the user logs in to a web application, they can access the pages which are designed for logged-in users. Also, depending on the user profile (authority level), the pages that are accessible to the user can be different. The key parts of user management function development are designing and building the process and pages.

4. User profile update

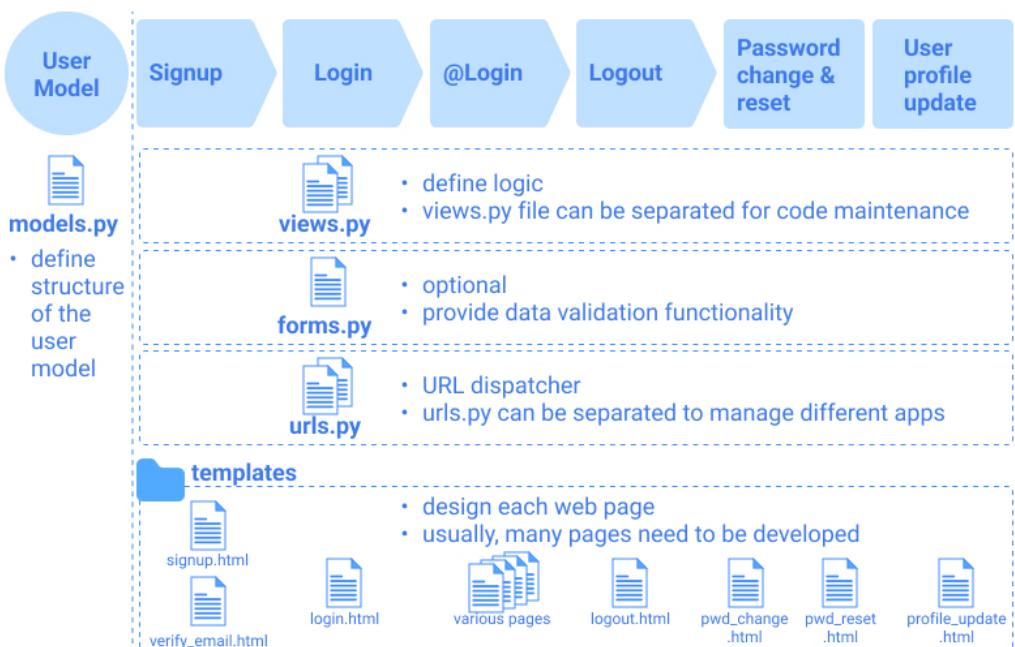
Usually, web applications have a page to manage user profiles. Unlike the primary user management functions, the user profile page design is more free-form.

5. Social login

The **social login** is an approach for user authentication via existing online service accounts such as Google, GitHub, or Facebook accounts. By using the social login feature, a web app's user management process can be simplified, and the users can avoid managing their passwords and user profiles. Social login is a common feature in many web applications.

User Management Function Development with Django

User Management Function Development with Django



The main figure describes key design points in user management function development using the Django framework.

1. models.py

`models.py` is used to define the structure of the user model. When you use the Django built-in user model, you don't need to add a new user model, but creating a custom model for the future expandability of the user model is recommended.

2. views.py

views.py is the core part of the logic that handles the process of user management. Usually, it is one *views.py* file per application, but you can separate files to make the coding structure more transparent. There are multiple approaches to developing views, which will be explained in the next section.

3. *forms.py*

forms.py is used to add **data validation** for input forms. When you use **class-based generic views**, you may not use this, but when you want to customize data validation, using *forms.py* is beneficial.

4. *urls.py*

urls.py is used to handle URLs to dispatch HTTP requests to related views. Usually, you need to create a *urls.py* file for user management (e.g., under the *account* app directory).

5. *templates*

The *templates* directory should contain multiple HTML files. User management-related templates are usually stored under the *account* sub-directory. As there are many steps for user management, you need to manage many template files (e.g., sign-up, login, log-out, and user profile pages).

Approaches to Building User Management Functions in Django

Approaches to Building User Management Functions in Django

Features and Key files	Approaches		
	1. Function-based view	2. Class-based view	3. django-allauth
Signup	*create_user(), set_password()	-	✓
Email Verification	-	-	✓
Login	*authenticate()	✓	✓
Logout	*logout()	✓	✓
Password Change	-	✓	✓
Password Reset	-	✓	✓
Social Login	-	-	✓
urls.py	-	✓	✓
Template files	-	Only default path for templates	✓

* related basic method examples ✓ available ready-made tools

There are three major approaches to building **user management** functions. The first approach is building them from scratch using **function-based views**. The second approach is using Django **built-in class-based views**. The last approach is leveraging Django libraries such as **django-allauth**.

1. Function-based views

Using function-based views, you can build user management functions from scratch. If you are implementing simple ones (for example, only signup, login, and logout), you can use this approach, but if you are planning to build more robust

functionality relating to the user management functions, it is better for you to try the other two approaches.

2. Class-based views

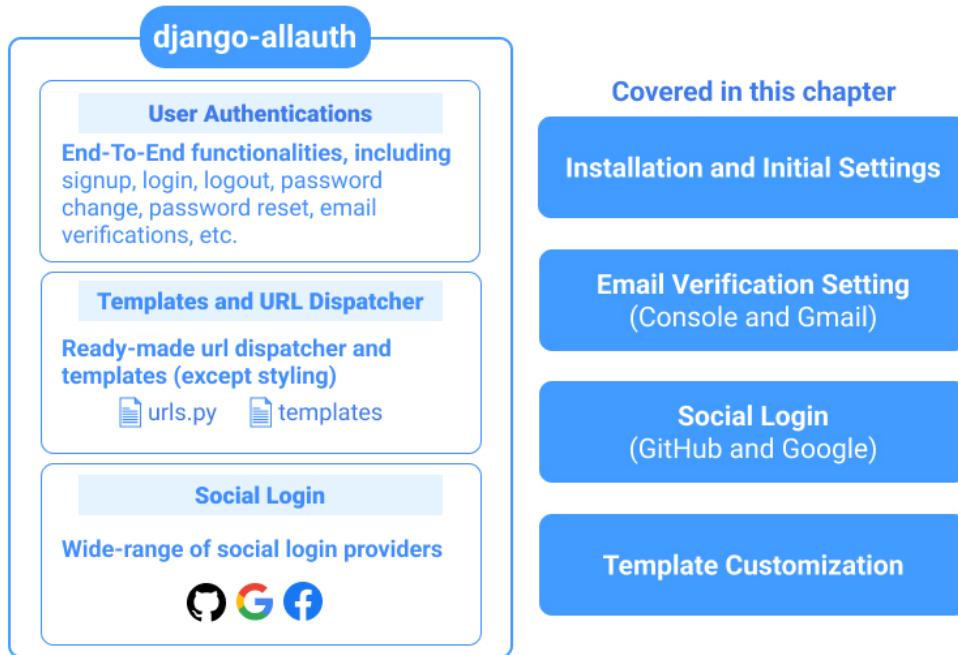
Django provides several built-in class-based views to build user management functions. Using the built-in class-based views, your development speed can be much faster than when using the function-based views.

3. Using a library (`django-allauth`)

There are Django libraries for user management. `django-allauth` is the most comprehensive library that provides a wide range of functionalities for user management, including views, forms, URL dispatchers, and many template files. It also offers **social login** functionality. In this chapter, we'll utilize `django-allauth` for quick implementation of key user management functions.

Django Allauth (1) – Introduction

Django Allauth (1) – Introduction



django-allauth is the most comprehensive library for Django authentication that provides a wide range of functionalities for user management, including **views**, **forms**, **URL dispatchers**, and many template files. It also offers **social login** functionalities.

In this chapter, we'll cover the following settings for django-allauth:

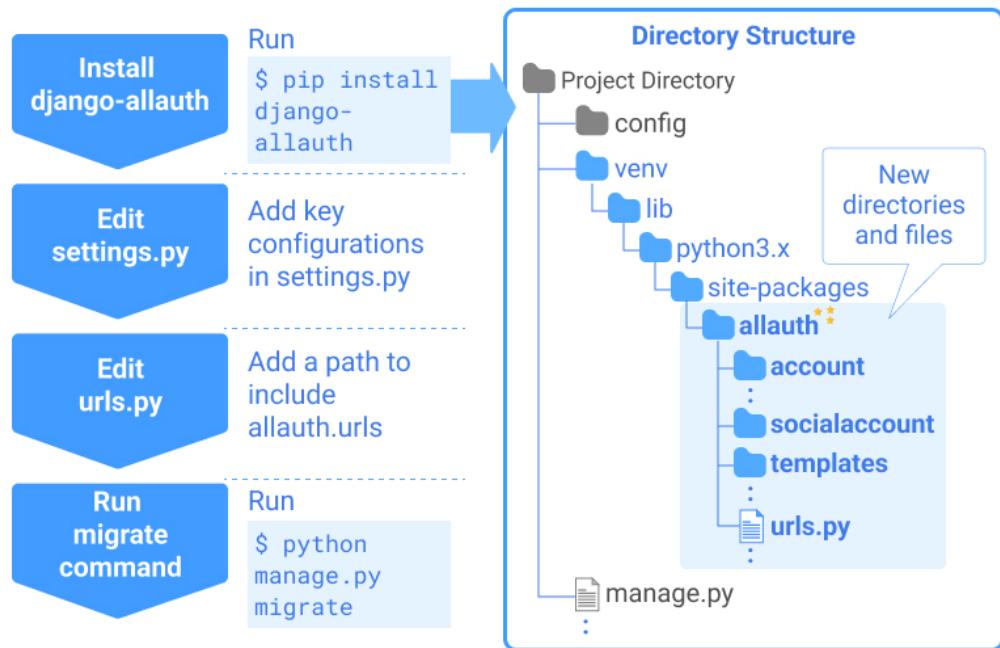
- **Installation and initial settings:** how to install django-allauth and how to make initial settings using email-based authentication
- **Email verification setting:** how to set up email verification using console (terminal) for a quick test in the development

environment; also covers how to set up email verification using Gmail.

- **Social Login:** how to set up social login for GitHub and Google account
- **Template Customization:** how to style template files using **Bootstrap** and **Crispy Forms**, including improving social login UI

Django Allauth (2) – Installation and Initial Settings

Django Allauth (2) – Installation and Initial Settings



The first steps to using **django-allauth** are installing it and adding initial settings. In this chapter, we continue to use the employee learning app to demonstrate django-allauth settings and functionalities.

1. Install django-allauth

To install django-allauth, add it to `requirements.txt`.

`requirements.txt`

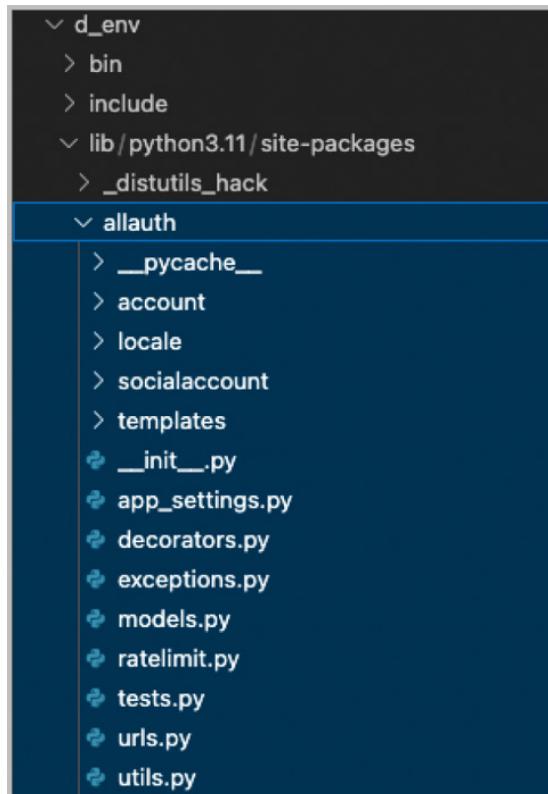
```
Django==4.1.7
django-crispy-forms
crispy-bootstrap5
django-allauth
```

Run the command below to install django-allauth.

Command Line - INPUT

```
(d_env) project_d % | pip install -r requirements.txt
```

Check the project directory tree. You can find that the *allauth* directory is created under *d_env/lib/python3.xx/site-packages* like in the image below.



The important file and directory are:

- The *urls.py* file under the *account* directory
- The *templates* directory located immediately under the *allauth* directory

The *urls.py* contains `urlpatterns` for user authentications. The *templates* directory contains many HTML templates that are used for user authentication UIs.

2. Edit settings.py

Add apps

The following apps are required according to the official document. Some apps are already registered. Add the ones in **yellow**.

config/settings.py

```
INSTALLED_APPS = [
    :
    'test_app',
    'employee_learning',
    'crispy_forms',
    'crispy_bootstrap5',
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
]
```

Add Site ID and Authentication Backends

Add the code below at the end of *settings.py*. To make the section in the *settings.py* file clear, you can add `### django-allauth settings###` in the beginning. The comments in green are the comments from the official documentation. The comments explain why we need these settings.

config/settings.py

```
### django-allauth settings ###

SITE_ID = 1

AUTHENTICATION_BACKENDS = [
    # Needed to login by username in Django admin, regardless of
    `allauth
    'django.contrib.auth.backends.ModelBackend',
    # `allauth` specific authentication methods, such as login by e-
    mail
    'allauth.account.auth_backends.AuthenticationBackend',
]
```

Login and logout redirection settings

Add the code below at the end. This code is used to set redirection URLs used for login and logout. We set the landing page for the **login redirect URL** and the login page for the **logout redirect URL**.

For now, comment out the `ACCOUNT_LOGOUT_ON_GET = True` part. This part will be explained later.

config/settings.py

```
'allauth.account.auth_backends.AuthenticationBackend',
]

# LOGIN and LOGOUT Redirection
LOGIN_REDIRECT_URL = '/'
ACCOUNT_LOGOUT_REDIRECT_URL = '/accounts/login/'
# ACCOUNT_LOGOUT_ON_GET = True
```

Change account authentication method to email

Lastly, add the code below. In this example, we'll use the email authentication approach, which has been recently more common than the username authentication approach. But for email verification, set it as "none" for now. We'll cover email verification settings later in this chapter.

config/settings.py

```
ACCOUNT_LOGOUT_REDIRECT_URL = '/accounts/login/'
# ACCOUNT_LOGOUT_ON_GET = True

ACCOUNT_AUTHENTICATION_METHOD = "email"
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_EMAIL_VERIFICATION = "none"

ACCOUNT_USERNAME_REQUIRED = False
ACCOUNT_UNIQUE_USERNAME = False

#Add the following when you are using custom user model
#ACCOUNT_USER_MODEL_USERNAME_FIELD = None
```

3. Edit urls.py

As you need to connect between the main *urls.py* and the *urls.py* under *allauth* directory, add a new path in the main *urls.py* shown below.

config/urls.py

```
urlpatterns = [
    ...
    path('accounts/', include('allauth.urls')),
]
```

The *urls.py* under the *allauth* directory is linked with another *urls.py* file under the *account* directory.

The `urls.py` file under the `account` directory contains the following `urlpatterns`. This list is important to understand how to check URLs and **URL names** used for user authentication pages. This is only for your reference. There is no need to edit this file.

allauth/account/urls.py

```
urlpatterns = [
    path("signup/", views.signup, name="account_signup"),
    path("login/", views.login, name="account_login"),
    path("logout/", views.logout, name="account_logout"),
    path("password/change/", views.password_change,
         name="account_change_password",),
    path("password/set/", views.password_set,
         name="account_set_password"),
    path("inactive/", views.account_inactive,
         name="account_inactive"),
    # E-mail
    path("email/", views.email, name="account_email"),
    path("confirm-email/", views.email_verification_sent,
         name="account_email_verification_sent",),
    re_path(r"^confirm-email/(?P[-:\w]+)/$", views.confirm_email,
            name="account_confirm_email",),
    # password reset
    path("password/reset/", views.password_reset,
         name="account_reset_password"),
    path("password/reset/done/", views.password_reset_done,
         name="account_reset_password_done",),
    re_path(r"^password/reset/key/(?P[0-9A-Za-z]+)-(?P.+)/$",
            views.password_reset_from_key,
            name="account_reset_password_from_key",),
    path("password/reset/key/done/",
            views.password_reset_from_key_done,
            name="account_reset_password_from_key_done",),
]
]
```

4. Run the migrate command

As migration files are already prepared, you just run the migrate command.

Command Line - INPUT

```
(d_env) project_d % | python manage.py migrate
```

Command Line - RESPONSE

```
Operations to perform:
  Apply all migrations: account, admin, auth, contenttypes,
  employee_learning, sessions, sites, socialaccount, test_app
Running migrations:
  Applying account.0001_initial... OK
```

```
Applying account.0002_email_max_length... OK
Applying sites.0001_initial... OK
Applying sites.0002_alter_domain_unique... OK
Applying socialaccount.0001_initial... OK
Applying socialaccount.0002_token_max_lengths... OK
Applying socialaccount.0003_extra_data_default_dict... OK
```

5. Check the results

To check if the initial settings are appropriately done, execute the `runserver` command.

Command Line - INPUT

```
d_env) project_d % | python manage.py runserver
```

The user authentication pages are available at the URL written in the `urls.py` file under `allauth/account`. Check one of the URLs, for example, `http://localhost:8000/accounts/login/`. Although UIs do not look good, you can access the key pages for user authentications now.

sign up first.' There are input fields for 'E-mail' and 'Password', a 'Remember Me' checkbox, and links for 'Forgot Password?' and 'Sign In'."/>

← → ⌂ ⓘ localhost:8000/accounts/login/

Menu:

- [Sign In](#)
- [Sign Up](#)

Sign In

If you have not created an account yet, then please [sign up](#) first.

E-mail:

Password:

Remember Me:

[Forgot Password?](#) [Sign In](#)

Try to sign in. You will be directed to the home page if you can successfully sign in.



Although you cannot see the login status, you are logged in to the app now. In this status, if you try to go to `http://localhost:8000/accounts/login/` again, you'll be redirected to the home page as it is set as a **login redirect URL**. If you want to log out, you need to type `http://localhost:8000/accounts/logout/`.

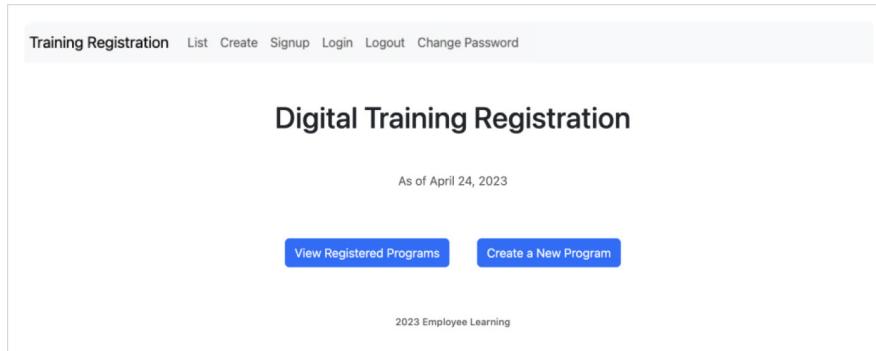
Continuing to type URLs is not efficient during the development process. Add links to Signup, Login, and Logout as a temporary solution in the navbar by editing the `base.html` file. Add the yellow part below. The URL names are picked from the `urls.py` file under `allauth/account`.

templates/base.html

```
<div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link" href="{% url 'course_list'%}">List</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'course_create'%}">Create</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'account_signup'%}">Signup</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'account_login'%}">Login</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'account_logout'%}">Logout</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{% url 'account_change_password'%}">Change Password</a>
        </li>
    </ul>
```

```
</div>
```

After saving the file, refresh the browser. You can now have access to the Logout page. Also, you can check that you cannot go to the Signup and Login page when you are in the login status.

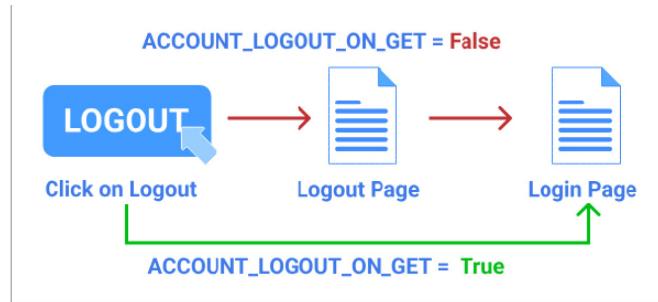


When you try to go to some pages, you may encounter the error like the one below. This is because we haven't done email-related settings, which will be explained in the next section.

The screenshot shows a Python exception traceback for a `ConnectionRefusedError` at the URL `/accounts/password/reset/`. The error message is `[Errno 61] Connection refused`. The traceback details the request method (POST), URL (`http://localhost:8000/accounts/password/reset/`), Django version (4.1.7), exception type (`ConnectionRefusedError`), exception value (`[Errno 61] Connection refused`), exception location (`/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/socket.py`, line 836, in `create_connection`), and raised during (`allauth.account.views.PasswordResetView`). It also shows the Python executable (`/Users/bloovee/Desktop/project_d/d_env/bin/python`), Python version (3.11.2), and Python path. The server time is listed as `Fri, 07 Apr 2023 14:32:12 +0000`. Below the exception, a "Traceback" section is shown with a link to "Switch to copy-and-paste view". The traceback shows the file `/Users/bloovee/Desktop/project_d/d_env/lib/python3.11/site-packages/django/core/handlers/exception.py`, line 56, in `inner`. The code line is `response = get_response(request)`. There is also a "Local vars" section.

Idea Note: Account Log On Get

In the Login and Logout redirection settings, we commented out the `"ACCOUNT_LOGOUT_ON_GET = True"` part. This setting is used to set the logout steps. If this is set as "True", the user is immediately logged out when they click on the link to the logout page.



The default setting is "`False`". Thus, when this setting is commented out, you still go to the Logout page when clicking on the logout link. To change the setting, take out `#` like shown below.

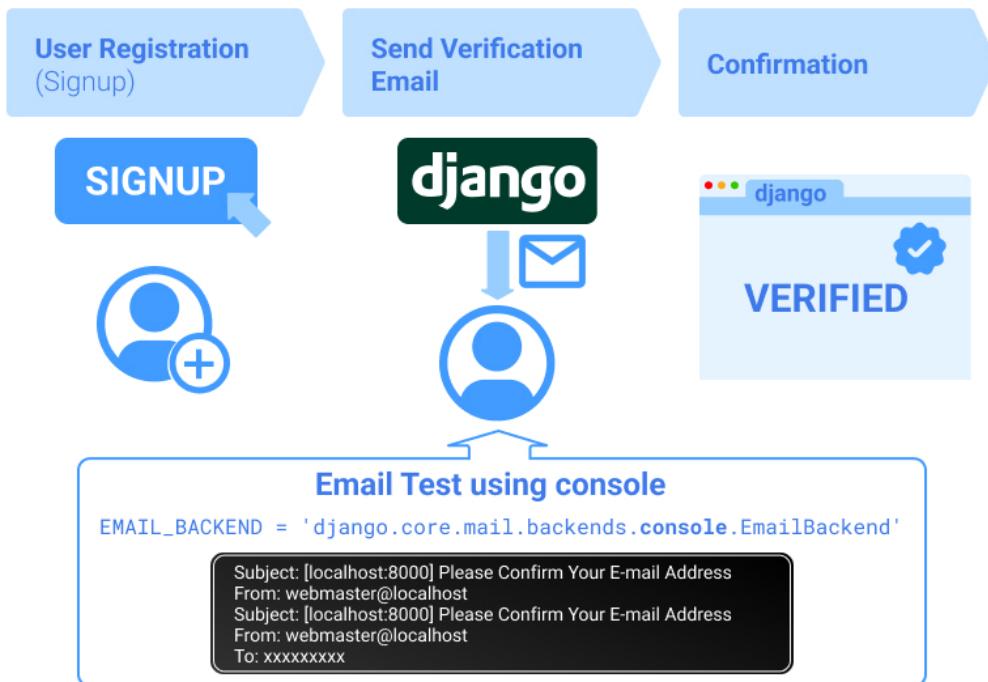
`config/settings.py`

```
# LOGIN and LOGOUT Redirection
LOGIN_REDIRECT_URL = '/employee-learning/course-list/'
ACCOUNT_LOGOUT_REDIRECT_URL = '/accounts/login/'
ACCOUNT_LOGOUT_ON_GET = True
```

Refresh the browser and click on the logout link. You'll be immediately logged out.

Django Allauth (3) – Email Verification via Console

Django Allauth (3) – Email Verification via Console



This lesson will explain how to set up email verification using a console (terminal) in the development environment.

1. Edit settings.py

The console-based email verification setting is easy to implement. You just need to edit the two lines of code in *settings.py*.

- Change `ACCOUNT_EMAIL_VERIFICATION` from "none" to "mandatory"
- Add `EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'` in the end of *settings.py*

The second setting enables Django to display email contents in the console.

config/settings.py

```
ACCOUNT_AUTHENTICATION_METHOD = "email"
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_EMAIL_VERIFICATION = "mandatory"
:
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

2. Check the results

To check the results, try to sign up again. You can use a dummy email, as no email will be actually sent.

Menu:

- [Sign In](#)
- [Sign Up](#)

Sign Up

Already have an account? Then please [sign in](#).

E-mail:

Password:

Password (again):

Once you hit the Sign Up button, you'll see the message below...

Messages:

- Confirmation e-mail sent to xxxx@gmail.com.

Menu:

- [Sign In](#)
- [Sign Up](#)

Verify Your E-mail Address

We have sent an e-mail to you for verification. Follow the link provided to finalize the signup process. If you do not see the verification e-mail in your main inbox, check your spam folder. Please contact us if you do not receive the verification e-mail within a few minutes.

...and you'll get a virtual email in your terminal like the one below. Go to the URL to see the result.

Command Line - RESPONSE

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: [example.com] Please Confirm Your E-mail Address
From: webmaster@localhost
To: xxxx@gmail.com
```

```
Date: Fri, 07 Apr 2023 14:53:30 -0000
Message-ID:
<168087921091.20140.5593477130948205989@macbook-
pro.myrepublic.com.sg>
Hello from example.com!

You're receiving this e-mail because user xxxx has given your
e-mail address to register an account on example.com.

To confirm this is correct, go to
http://localhost:8000/accounts/confirm-
emailMTY:1pknT0:wDqaqGgeBGx6IMKF76NQU9n8tyVsrqpJC1fXFtKJ9fE/

Thank you for using example.com!
example.com
```

On the browser, you'll find the email confirmation message. Once you press the **Confirm** button, your account will be created...

Menu:

- [Sign In](#)
- [Sign Up](#)

Confirm E-mail Address

Please confirm that xxxxx@gmail.com is an e-mail address for user xxxx.

...and you'll be ready to sign in (log in) to the service.

Messages:

- You have confirmed xxxx@gmail.com.

Menu:

- [Sign In](#)
- [Sign Up](#)

Sign In

If you have not created an account yet, then please [sign up](#) first.

E-mail:

Password:

Remember Me:

[Forgot Password?](#)

Sign in with your registered email and password. You'll jump to the home page (You are successfully logged in to the app).

Training Registration [List](#) [Create](#) [Signup](#) [Login](#) [Logout](#) [Change Password](#)

Digital Training Registration

As of April 24, 2023

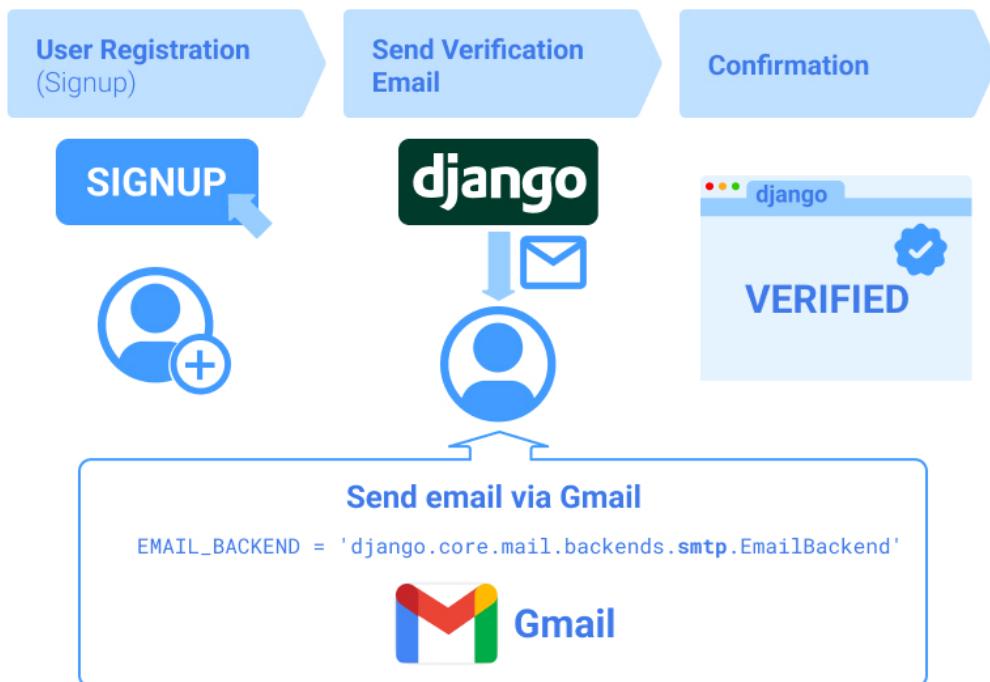
[View Registered Programs](#)

[Create a New Program](#)

2023 Employee Learning

Django Allauth (4) – Email Verification via Gmail

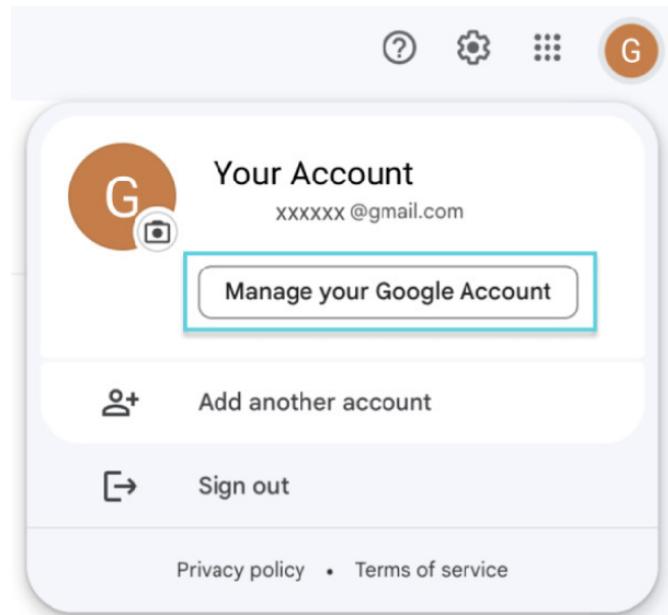
Django Allauth (4) – Email Verification via Gmail



This lesson will explain how to set up email verification via Gmail. The approach can change depending on Google's security policy. This approach is valid as of April 2023.

1. Set up a Gmail App password

You need to set up a Gmail App password to send email verification emails via Gmail. Go to your Gmail account in your browser and click on "**Manage your Google Account**".



Go to **2-Step Verification** under the **Security** section.

A screenshot of the Google Security settings page. On the left, a sidebar lists 'Home', 'Personal info', 'Data and privacy' (which is selected and highlighted with a blue box), 'Security' (also highlighted with a blue box), 'People and sharing', 'Payments and subscriptions', and 'About'. The main content area is titled 'Security' with the subtitle 'Settings and recommendations to help you keep your account secure'. It features three sections: 'You have security tips' (with a 'Review security tips' link), 'Recent security activity' (stating 'No security activity or alerts in the last 28 days'), and 'How you sign in to Google' (with options for '2-Step Verification' and 'Use your phone to sign in'). The '2-Step Verification' option is also highlighted with a blue box.

If you haven't set it up yet, follow the guidance to set up **2-Step Verification** and turn it on.

[←](#) 2-Step Verification



Protect your account with 2-Step Verification

Prevent hackers from accessing your account with an additional layer of security. When you sign in, 2-Step Verification helps make sure that your personal information stays private, safe and secure.



Security made easy

In addition to your password, 2-Step Verification adds a quick, second step to verify that it's you.



Use 2-Step Verification for all your online accounts

2-Step Verification is a proven way to prevent widespread cyber-attacks. Turn it on wherever it's offered to protect all your online accounts.

 Safer with Google

[Get started](#)

Go to the **App passwords** section at the bottom.

2-Step Verification

Authenticator app
Use the authenticator app to get verification codes at no charge, even when your phone is offline. Available for Android and iPhone.

Security Key
A security key is a verification method that allows you to securely sign in. These can be built in to your phone, use Bluetooth or plug directly into your computer's USB port.

Devices that don't need a second step
You can skip the second step on devices that you trust, such as your own computer.

Devices you trust
Revoke trusted status from your devices that skip 2-Step Verification.

[REVOKE ALL](#)

App passwords
App passwords aren't recommended and are unnecessary in most cases. To help keep your account secure, use 'Sign in with Google' to connect apps to your Google Account.

App passwords
None

Generate an app password with a custom name.

App passwords

App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it. [Learn more](#)

You don't have any app passwords.

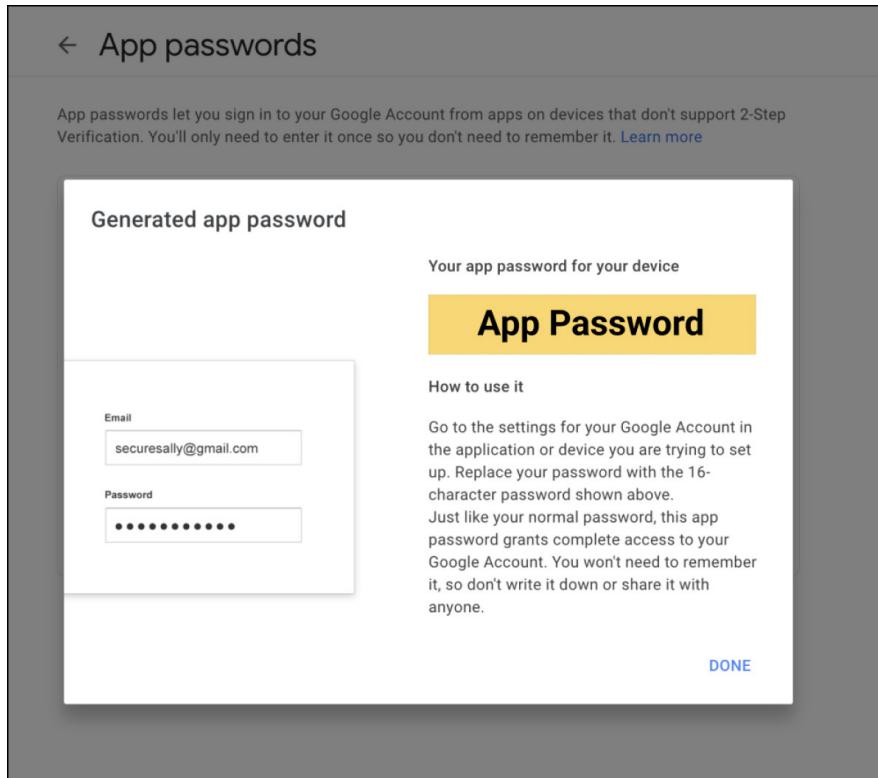
Select the app and device for which you want to generate the app password.

Select app: Mail, Calendar, Contacts, YouTube, Other (Custom name)

Select device: ▾

[GENERATE](#)

You'll see that an app password was generated. Copy the password as we will use it in `settings.py`.



2. Edit settings.py

To change the console-based email verification setting to email-based verification, edit *settings.py*.

- Change `EMAIL_BACKEND` to '`'django.core.mail.backends.smtp.EmailBackend'`'
- Add email settings

The `EMAIL_BACKEND` setting enables Django to send emails using the **SMTP protocol**.

config/settings.py

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'App Password'
EMAIL_USE_SSL = True
EMAIL_USE_TLS = False
EMAIL_PORT = 465
DEFAULT_FROM_EMAIL = 'your_account@gmail.com'
```

Make sure that only one of `TLS` or `SSL` can be True, and that the port setting is aligned with the protocol. In this case, we use `EMAIL_USE_SSL = True`, `EMAIL_USE_TLS = False`, and `EMAIL_PORT = 465`.

3. Check the results

To check the results, try to sign up again. This time, you need to use an actual email address.

Menu:

- [Sign In](#)
- [Sign Up](#)

Sign Up

Already have an account? Then please [sign in](#).

E-mail:

Password:

Password (again):

[Sign Up »](#)

Once you click on the **Sign Up** button, you'll see a message saying that a verification email has been sent out.

Messages:

- Confirmation e-mail sent to xxxx@gmail.com.

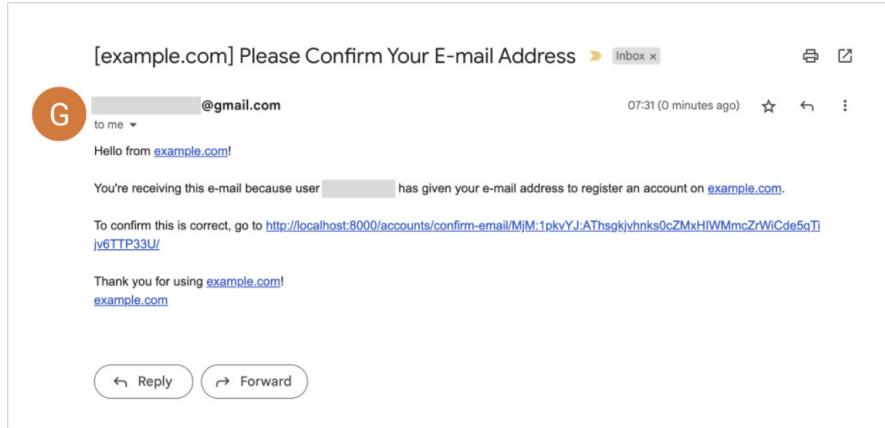
Menu:

- [Sign In](#)
- [Sign Up](#)

Verify Your E-mail Address

We have sent an e-mail to you for verification. Follow the link provided to finalize the signup process. If you do not see the verification e-mail in your main inbox, check your spam folder. Please contact us if you do not receive the verification e-mail within a few minutes.

Check your email. You'll see the verification email in your inbox.



Click the link, and then you'll see a confirmation message. After you press the **Confirm button**, your account will be created...

Menu:

- [Sign In](#)
- [Sign Up](#)

Confirm E-mail Address

Please confirm that [xxxxx@gmail.com](#) is an e-mail address for user xxxx.

...and you'll be ready to sign in (log in) to the service.

Messages:

- You have confirmed xxxx@gmail.com.

Menu:

- [Sign In](#)
- [Sign Up](#)

Sign In

If you have not created an account yet, then please [sign up](#) first.

E-mail:

Password:

Remember Me:

[Forgot Password?](#)

Sign in with your registered email and password. You'll jump to the home page (You are successfully logged in to the app).

Training Registration List Create Signup Login Logout Change Password

Digital Training Registration

As of April 24, 2023

[View Registered Programs](#) [Create a New Program](#)

2023 Employee Learning

Note: Delete Users

When you are working on the email setting, you may want to test functionalities multiple times. However, the app blocks new registrations that use the same email address. Delete the user from the Django admin to test it again with the same email address.

Go to the **Users** section on the left menu. Select the user to delete, select "**Delete selected users**" in the dropdown, and press the **Go** button.

Select user to change

Action:	Go	1 of 2 selected	
Delete selected users			
EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
bloovee	bloovee@example.com		✓
xxxxx	xxxxx@gmail.com		✗

2 users

Make sure that you delete the user, not just their email address; or else, even though you delete the email address, the user won't be deleted. But once you delete the user, the registered email address will also be deleted.

Django Allauth (5) – Social Login with GitHub

This lesson will explain how to set up a social login with GitHub. The approach can change depending on GitHub's policy. This approach is valid as of April 2023.

1. Register a new OAuth app on the GitHub website

Go to your [GitHub](#) account page to set up a new OAuth app. Open the menu from the user icon on the top right. Select Settings.

Select Developer settings on the left sidebar.

Clicks on the Register a new application button under the OAuth Apps menu.

There are three items you need to type in.

- App Name: Employee Learning (any name)
- Homepage URL: `http://localhost:8000`
- Authorization callback URL:
`http://localhost:8000/accounts/github/login/callback/`

You need to type the Homepage URL and callback URL carefully. Often, errors come from this setting. If you are using `127.0.0.1` for the runserver command, use `127.0.0.1` here as well. Another point you need to be careful about is the accounts part. Don't forget to put `s` at the end.

Once you type them correctly, press the Register application button.

You can see the registered app details with Client ID. Click on the Generate a new client secret button to generate a Client secret. Copy the Client ID and Client secret, and save them in a safe place. We'll use them later.

2. Edit settings.py

To add the GitHub social login feature, you need to edit settings.py.

- Add allauth github app in the INSTALLED_APPS section
- Add SCOPE under SOCIALACCOUNT_PROVIDERS

config/settings.py

```
INSTALLED_APPS = [  
:  
'allauth.socialaccount.providers.github',  
]  
###Social Login###
```

```
SOCIALACCOUNT_PROVIDERS = {
```

```
'github': {
```

```
'SCOPE': [  
'user',
```

```
'repo',
```

```
'read:org',
```

```
],
```

```
},
```

```
}
```

You may need to add your server URL if you are using GitHub Enterprise. Check the [allauth official document](#).

3. Register in Django Admin

As the last step, you need to add your domain on the Site page and GitHub social login settings on the Social applications page in the Django admin site.

Update the domain information on the Site page

Click on the Site link on the left sidebar. As a default, example.com is registered on the Site page.

Change it to localhost:8000.

Add GitHub social login settings on the Social applications page

Click on the +Add button beside the Social applications link on the left sidebar.

Add the following settings:

- Provider: GitHub
- Name: Any name

- Client ID: Client ID obtained from GitHub website
- Secret Key: Client secret obtained from GitHub website
- Sites: localhost:8000

Once it's done, save the record.

Once it's done, save the record.

4. Check the results

Go to the Sign In (Login) page (not the Sign Up page!) to check the results.

You can see the link named GitHub.

Click on the link. And, then, click on the Continue button.

You'll be redirected to the GitHub site. Press the Authorize button.

Confirm the access with GitHub Mobile or password.

If you are using GitHub Mobile, type the code on your mobile

Once access is confirmed, you'll be redirected to the Sign Up page.

Once access is confirmed, you'll be redirected to the Sign Up page.

Once you click on the Sign Up button, you'll see a message saying that a verification email has been sent out.

Check your email. You'll see the verification email in your inbox.

Click the link, then you'll see a confirmation message. After you press the Confirm button, your account will be created...

...and you'll be ready to sign in (log in) using GitHub Social Login.

Click on the GitHub link. Then, click on the Continue button.

You'll jump to the landing page (You are successfully logged in to the app).

Note: Revoke Access

When a user wants to revoke the GitHub social login for your web application, go to Applications under Settings. And select Authorized OAuth Apps.

Select the app that the user wants to revoke, and click on the Revoke access button.

When the user wants to use the GitHub social login for your web application again, they will need to reauthorize the access through your web application.

Django Allauth (6) – Social Login with Google

This lesson will explain how to set up a social login with Google. The approach can change depending on Google's policy. This approach is valid as of April 2023.

1. Register a new OAuth app in the GCP (Google Cloud Platform) console

Go to the [GCP \(Google Cloud Platform\)](#). Click on the Console button on the top bar.

Click on Select a project. And press the NEW PROJECT button.

Add a project name (e.g., Employee Learning) and press the CREATE button.

In the created project (Employee Learning), select API and services. Click on OAuth consent screen.

On the OAuth consent screen, select External and press the CREATE button.

Add App name, User support email, and Developer contact information at the bottom.

There is no need to add other information. Click on Save And Continue in the rest of the steps.

Once the OAuth consent part is done, select Credentials on the left sidebar. And click on + CREATE CREDENTIALS. Select OAuth client ID from the dropdown menu.

Add the following information.

- Application Type: Web application
- Name: Any name (e.g., Employee Learning)
- Authorized JavaScript origins: http://localhost:8000
- Authorized redirect URIs:
http://localhost:8000/accounts/google/login/callback/

Carefully type, especially for Authorized JavaScript origins and Authorized redirect URIs.

- Authorized JavaScript origins is the Homepage URL in GitHub.
- Authorized redirect URIs is Authorization callback URL in GitHub.

Add the following information.

Often, error come from this setting. If you are using 127.0.0.1 for the runserver command, use 127.0.0.1 here as well. Another point you need to be careful about is the accounts part. Don't forget to put s at the end.

Once you have filled in all the required data, click on CREATE. You can see a pop-up with the Client ID and Client secret and save them somewhere in a safe place. We'll use them later.

2. Edit settings.py

To add the Google social login feature, you need to edit settings.py.

- Add allauth google app in the INSTALLED_APPS section
- Add SCOPE and other settings under SOCIALACCOUNT_PROVIDERS

config/settings.py

```
INSTALLED_APPS = [  
    :  
    'allauth.socialaccount.providers.github',  
    'allauth.socialaccount.providers.google',  
]
```

###Social Login###

```
SOCIALACCOUNT_PROVIDERS = {  
    'github': {  
        'SCOPE': [  
            'user',
```

```
'repo',
'read:org',
],
},
'google': {
'SCOPE': [
'profile',
'email',
],
'AUTH_PARAMS': {
'access_type': 'online',
},
'OAUTH_PKCE_ENABLED': True,
},
}
```

Check the [allauth official document](#) for details.

3. Register in Django Admin

As the last step, you need to add your domain on the Site page and Google social login settings on the Social applications page in the Django admin site.

Update the domain information on the Site page

You can skip this step if you have done this in the previous GitHub section.

Click on the Site link on the left sidebar. As a default, example.com is registered on the Site page.

Change it to localhost:8000.

Add Google social login settings on the Social applications page

Click on the +Add button beside the Social applications link on the left sidebar.

Add the following settings:

- Provider: GitHub
- Name: Any name
- Client ID: Client ID obtained from GitHub website
- Secret Key: Client secret obtained from GitHub website
- Sites: localhost:8000

Once it's done, save the record.

Once it's done, save the record.

4. Check the results

Go to the Sign In (Login) page (not the Sign Up page!) to check the results.

You can see the link named Google.

Click on the link. And then click on the continue button.

You'll be redirected to the Google's site. Select an account you want to use for the social login.

Once you select an account, you'll jump to the landing page (You are successfully logged in to the app).

Unlike with the GitHub social login setting, you don't need to verify your email again.

Django Allauth (7) – Allauth Template File Setup

This lesson will explain how to edit template files for allauth.

Create mirror template files in the project templates directory

As you can see in the main figure, template files for allauth are located in the templates directory under the allauth directory. Directly editing these original template files is not a good idea. You should copy all the directories and paste them into the project templates directory immediately under the project directory.

By doing this, Django will read the template files as if those files were located in the original location. If you want to change the design of the user authentication pages, edit the copied html files.

Right-click on VS Code

For quick operation, in VS Code, you can use the right-click to copy and paste the directories.

Test

To test if everything is working, add the following one-line code in the signup.html file under the account directory.

After saving the file and making sure the development server (runserver) is running, go to the Sign Up page. You can see that the added code is displayed like shown in the screenshot below.

As this is only for testing purposes, delete the test code and save the file.

Now you are ready to edit the template files for allauth.

Django Allauth (8) – Add Basic Styling with Bootstrap and Crispy Forms

This lesson will explain how to add basic styling to the allauth pages quickly.

1. Add key modules in account/base.html

The template files for allauth extend the base.html file under the account directory. Thus, adding styles in this file can speed up the entire UI design formatting.

Add the three tags below in the base.html file under the account directory. You can quickly include css_link.html, navbar.html, and footer.html.

- {%- include "base/css_link.html" %}
- {%- include "base/navbar.html" %}
- {%- include "base/footer.html" %}

templates/account/base.html

```
{% load i18n %}

<!DOCTYPE html>

<html>

: 

<meta name="viewport" content="width=device-width, initial-scale=1.0">

{%- include "base/css_link.html" %}

<title>{%- block head_title %}{%- endblock %}</title>
```

```
:  
  
<body>  
{> include "base/navbar.html" %}  
{> block body %}  
  
:  
  
{> endblock %}  
{> include "base/footer.html" %}  
  
</body>  
  
</html>
```

2. Customize the navigation bar

The following part of the code in the account/base.html file is designed for the navigation bar. We'll utilize these codes in our navigation bar.

```
templates/account/base.html  
:  
  
<div>  
  
<strong>{> trans "Menu:" %}</strong>  
  
<ul>  
  
{> if user.is_authenticated %}  
  
<li><a href="{> url 'account_email' %}">{> trans "Change E-mail" %}  
</a></li>
```

```
<li><a href="{% url 'account_logout' %}">{% trans "Sign Out" %}</a>
</li>

{% else %}

<li><a href="{% url 'account_login' %}">{% trans "Sign In" %}</a></li>

<li><a href="{% url 'account_signup' %}">{% trans "Sign Up" %}</a>
</li>

{% endif %}

</ul>

</div>

:
```

Cut the code above and paste it into navbar.html. And replace the existing tags for authentications.

Delete unnecessary parts (e.g., menu), and add classes "nav-item" and "nav-link" to the new tags.

Also, delete {% trans %} tags.

templates/base/navbar.html

```
<ul class="navbar-nav">

<li class="nav-item">

<a class="nav-link" href="{% url 'course_list'%}">List</a>

</li>

li class="nav-item">

<a class="nav-link" href="{% url 'course_create'%}">Create</a>
```

```
</li>

{% if user.is_authenticated %}

<li class="nav-item">

<a class="nav-link" href="{% url 'account_email' %}">Change E-mail</a>

</li>

<li class="nav-item">

<a class="nav-link" href="{% url 'account_logout' %}">Sign Out</a>

</li>

{% else %}

<li class="nav-item">

<a class="nav-link" href="{% url 'account_login' %}">Sign In</a>

</li>

<li class="nav-item">

<a class="nav-link" href="{% url 'account_signup' %}">Sign Up</a>

</li>

{% endif %}

</ul>
```

By implementing the above, the navbar menu changes depending on the login status, like shown in the images below.

3. Add quick styling

Apply the following four styling approaches in all the relevant template files.

Overall layout

In the account/base.html file, add a new container tag <div class="container w-50" style="min-width:300px; max-width: 500px;"> to wrap all elements
templates/account/base.html

:

{% endif %}

<div class="container w-50" style="min-width:300px; max-width: 500px;">

{% block content %}

{% endblock %}

{% endblock %}

{% block extra_body %}

{% endblock %}

</div>

{% include "base/footer.html" %}

:

Doing this will move the main content to the center.

Title <h1> tag

In the custom.css file, add the following style for <h1> tag

- text-align: center;
- margin: 3rem;

This gives you the same results as with class="m-5 text-center", which we used for the list page, etc.

static/css/custom.css

```
body {  
background-color: white;  
}  
  
h1{  
text-align: center;  
margin: 3rem;  
}
```

Doing this will make the positioning of the title more appealing.

Buttons

You can add class="btn btn-primary d-block mx-auto" to style buttons. In some cases, you can also change color using "warning" or "danger" etc. To do that, check all HTML files under the account and socialaccount directory and find the <button> and <input type="submit"> tags.

If there is a class in the tag already, add the "btn btn-primary d-block mx-auto" part in the tag. Browsers may not read the additional class properly.

For example, the second part (the blue part) of the class in the following code may not be processed in browsers properly.

templates/account/login.html

```
<button class="primaryAction" class="btn btn-primary d-block mx-auto"  
type="submit">{ % trans "Sign In" %}</button>
```

Instead, you need to change to the yellow part below.

templates/account/login.html

```
<button class="primaryAction btn btn-primary d-block mx-auto"  
type="submit">{ % trans "Sign In" %}</button>
```

Now, the buttons look better.

In some cases, you may need to put in extra effort into styling. For example, the Change E-mail page below looks ugly.

To fix this, use Bootstrap class and style tag to adjust color, button size, font size, and other properties.

templates/account/email.html

```
<div class="buttonHolder text-center">  
  
<button class="secondaryAction btn btn-primary w-75 m-1 mt-4"  
type="submit" name="action_primary" style="height: 2.5rem; border-  
radius:10px;">{ % trans 'Make Primary' %}</button>
```

```
<button class="secondaryAction btn btn-warning w-75 m-1" type="submit"
name="action_send" style="height: 2.5rem; border-radius:10px;">{%
trans 'Re-send Verification' %}</button>

<button class="primaryAction btn btn-danger w-75 m-1" type="submit"
name="action_remove" style="height: 2.5rem; border-radius:10px;">{%
trans 'Remove' %}</button>

</div>
```

The page looks better now.

Forms

Finally, we adjust forms using Crispy Forms. Add the {% load crispy_forms_tags %} and {{ form|crispy }} tags on the relevant pages.

For each html file under the account and social account directory, search "{{ form.as_p }}".

The code below is an example of the Change E-mail page. We also added some styling specifically for this page.

templates/account/email.html

```
{% if can_add_email %}

<hr>

<h2 class="m-5 text-center">{%
trans "Add E-mail Address" %}</h2>

{% load crispy_forms_tags %}

<form method="post" action="{%
url 'account_email' %}"
class="add_email">
```

```
{% csrf_token %}

{{ form|crispy }}

<button name="action_add" type="submit" class="btn btn-primary d-block mx-auto">{% trans "Add E-mail" %}</button>

</form>

{% endif %}

{% endblock %}
```

Now the page looks like this.

Note: Super Reload

Although you edited the html file and saved it, your browser may display the old content. This is because the browser may not recognize if there were changes in the html file. And, the browser renders the old content using cache.

To make the browser refresh the content to the new content, you need to super-reload the content. Usually, you need to click on the browser refresh button while pressing the Shift key on your keyboard.

Django Allauth (9) – Customize Sign-in and Sign-up Pages

The last part of the social login UI styling adjustment is for the Sign In (login) and Sign Up page. Currently, the social login UI design is just simple text, and social login is only available on the Sign In (login) page. In this section, we'll restructure the pages and add logos.

Add logos

As the links to the social login (e.g., GitHub and Google) are coming from the allauth source code, you may have difficulty replacing text links with image links. Here are the steps to add images to the social login links.

Prepare icons

We use these Google and GitHub icons for the links. To make them accessible by Django, save these image files under the images directory.

Define HTML elements to add icons

First, you need to identify the elements where you want to add the icons. This part may be the most challenging part.

To see where the GitHub's and Google's texts come from, check the account/login.html file. You can see that this HTML document includes another HTML document – socialaccount/snippets/provider_list.html.

templates/account/login.html

{% if socialaccount_providers %}

<p>{ % blocktrans with site.name as site_name %}Please sign in with one of your existing third party accounts. Or, sign up

for a {{ site_name }} account and sign in below:<% endblocktrans %></p>

```
<div class="socialaccount_ballot">
<ul class="socialaccount_providers">
    { % include "socialaccount/snippets/provider_list.html" with
        process="login" %}
</ul>

<div class="login-or">{ % trans 'or' %}</div>
</div>

{ % include "socialaccount/snippets/login_extra.html" %}

{ % else %}

<p>{ % blocktrans %}If you have not created an account yet, then please
<a href="{{ signup_url }}>sign up</a> first.<% endblocktrans %></p>

{ % endif %}
```

Check the socialaccount/snippets/provider_list.html file to identify the location where the text data is coming from.

```
templates/socialaccount/snippets/provider_list.html

{ % load socialaccount %}

{ % get_providers as socialaccount_providers %}
```

```

{%- for provider in socialaccount_providers %}

{%- if provider.id == "openid" %}|
```

{% for brand in provider.get_brands %}

```

<li>
```

```

<a title="{{brand.name}}" class="socialaccount_provider {{provider.id}} {{brand.id}}" href="{% provider_login_url provider.id
openid=brand.openid_url process=process %}">{{brand.name}}</a>
```

```

</li>
```

{% endfor %}

```

{% endif %}
```

```

<li>
```

```

<a title="{{provider.name}}" class="socialaccount_provider
{{provider.id}}" href="{% provider_login_url provider.id process=process
scope=scope auth_params=auth_params %}">{{provider.name}}</a>
```

```

</li>
```

{% endfor %}

The text data comes from {{brand.name}} or {{provider.name}}. We found the location of the HTML code; however, we cannot add links to the images as {{brand.name}} and {{provider.name}} are variables.

To find more information, go to the browser and select "inspect" with a right-click.

Now, you can see the actual code implemented in a browser. The GitHub and Google elements have their own classes – github and google. We can utilize these class names to identify the location of the elements where we want to add images.

Add images to the elements

We can utilize the custom.css file to add images. In this example, we'll use the background-image property to show the icons. The path is the relative path from the CSS file to the image file. As this file is not a Django template, you don't need to use the {% url %} tag. Add several properties to adequately adjust the size of the icons.

```
static/css/custom.css
```

```
.github {  
    display: inline-block;  
    content: " ";  
    width: 50px;  
    height: 50px;  
    background-image: url("../images/github.jpg");  
    background-size: contain;  
}  
  
.google {  
    display: inline-block;  
    content: " ";  
    width: 50px;  
    height: 50px;  
    background-image: url("../images/google.jpg");  
}
```

```
background-size: contain;
```

```
}
```

At this point, the icons are overlapping with the text. Also, there are list markers, and the icons are vertically listed.

To fix the page, edit socialaccount/snippets/provider_list.html by doing the following:

- Add d-flex to arrange the icons horizontally
- Delete the markers using the list-style-type property
- Delete {{provider.name}}

```
templates/socialaccount/snippets/provider_list.html
```

```
{% load socialaccount %}
```

```
{% get_providers as socialaccount_providers %}
```

```
<div class="d-flex justify-content-center gap-3">
```

```
{% for provider in socialaccount_providers %}
```

```
{% if provider.id == "openid" %}
```

```
{% for brand in provider.get_brands %}
```

```
<li>
```

```
<a title="{{brand.name}}"
```

```
class="socialaccount_provider {{provider.id}} {{brand.id}}"
```

```
href="{% provider_login_url provider.id openid=brand.openid_url
process=process %}"  
>{{brand.name}}</a>  
</li>  
{% endfor %}  
{% endif %}  
  
<li style="list-style-type: none;">  
<a title="{{provider.name}}" class="socialaccount_provider  
{% provider.id %}"  
href="{% provider_login_url provider.id process=process scope=scope  
auth_params=auth_params %}">###delete###</a>  
</li>  
{% endfor %}  
</div>
```

Now you can see a better design.

Adjust the layout of the Sign In and Sign Up pages

Typically, social login comes after the standard login approach. Also, social login should be available on the Sign Up page as well. In this part, we'll skip the detailed explanations as several minor adjustments are needed. Here are the high-level guidance and results.

Sign In page

Make the following adjustments:

- Move up the standard login code before the social login code
- Move the Forgot Password link to the right
- Move the sign up link after the Sign In button
- Add a horizontal line after the sign up link
- Simplify the message for the social login
- Adjust font sizes

Now you can see a much cleaner Sign In page.

Sign Up page

Make the following adjustments:

- Bring a social login code from the Sign In page
- Add the {load account socialaccount%} tag to make the social login code function
- Adjust font sizes

Now you can see a much cleaner Sign Up page.

User Models

Django provides a built-in user model with predefined fields and methods, including the following fields.

- username
- first_name
- last_name
- email
- password
- groups
- user_permissions
- is_staff
- is_active
- is_superuser
- last_login
- date_joined

You may want to add or select the required fields to the app you are developing. There are two primary ways to do it.

1. Extending the user model
2. Substituting the user model

1. Extending the user model – Profile model with OneToOne relationship

In your app, you may want to add more fields on top of the Django built-in user model, such as a user icon, mobile number, and other user profiles. Unless you intend to delete or modify a user authentication method, extending the user model may be easier.

The approach is to create an extended model using OneToOneField. Create a new model (e.g., the UserProfile model) with a One-To-One relationship with the built-in user model.

Practice 1

Objective: **Extending the User Model (User Profile)**

In this practice, we'll demonstrate how to extend the built-in user model by making a new UserProfile model that has a One-To-One relationship with the built-in user model.

1. Create a new app

As the UserProfile model is not related to any of the models under the employee_learning app, it is better to create a new app by running the startapp command. We use 'users' as an app name this time.

Command Line - INPUT

```
d_env) project_d % | python manage.py startapp user_profile
```

2. Add the app in settings.py

As the app is new, add it under INSTALLED_APPS in settings.py.

config/settings.py

```
INSTALLED_APPS = [
```

```
:
```

```
'user_profile',
```

```
:
```

```
]
```

3. Add the UserProfile model in models.py

Open models.py under the new app, and add the code below.

```
user_profile/models.py?
```

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

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

```
class UserProfile (models.Model):
```

```
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

```
    mobile = models.CharField(max_length=15, blank=True)
```

```
    def __str__(self):
```

```
        return self.user.username
```

4. Edit admin.py

To add the UserProfile model to the admin page, edit the admin.py file under the users directory.

```
users/admin.py
```

```
from django.contrib import admin
```

```
from .models import UserProfile  
admin.site.register(UserProfile)
```

5. Check the results

Run the makemigrations and migrate commands as we have created a new model. To see the outcome, execute the runserver command.

Command Line - INPUT

```
d_env) project_d % | python manage.py makemigrations user_profile  
d_env) project_d % | python manage.py migrate  
d_env) project_d % | python manage.py runserver
```

Go to localhost:8000/admin. You can see that the UserProfile model is registered under the new app. Using the same approach, you can extend the user model further.

Substituting the user model – Creating a custom model using AbstractUser

If you want to have a more customized user model with specific fields, you need to build a custom model inheriting the built-in user model.

AbstractUser vs. AbstractBaseUser

There are two built-in user models that you can utilize – AbstractUser and AbstractBaseUser.

AbstractUser

As AbstractUser is designed based on AbstractBaseUser, more fields and attributes are available. If you want to make a custom user model quickly, inheriting AbstractUser is better. However, this approach may be less flexible as there may be fields that you may not want to include in your custom user model (e.g., first_name and last_name).

AbstractBaseUser

AbstractBaseUser provides only the core implementation of a user model. Using AbstractBaseUser, you can build a new custom user model almost from scratch. If you want to design your custom user model in a more flexible way, it is better to inherit AbstractBaseUser. However, you need to keep in mind that more code is required to implement a new custom user model.

Practice 2

Objective: **Substituting the User Model**

In this practice, we'll demonstrate how to develop a custom user model by inheriting AbstractUser. As we have migrated the user model to the employee learning app, we'll generate a new project for this practice.

1. Start a new project

To create a new project, open a new window in VS Code. Create a new folder named 'custom_user', and open Terminal in VS Code. Then, run the following commands to create a virtual environment and prepare for Django installation.

Command Line - INPUT

```
customer_user % | python3 -m venv venv  
customer_user % | source venv/bin/activate
```

Command Line - INPUT

```
(venv)customer_user % | touch requirements.txt
```

Edit the requirements.txt file in VS Code by adding Django with version information.

requirements.txt

Django==4.1.7

Execute pip to install Django and run the startproject command.

Command Line - INPUT

```
(venv)customer_user % | python -m pip install --upgrade pip
```

```
(venv)customer_user % | pip install -r requirements.txt
```

```
(venv)customer_user % | django-admin startproject config .
```

Run the runserver command.

Command Line - INPUT

```
(venv)customer_user % | python manage.py runserver
```

Check localhost:8000. If you can see the following screen, your project start is successful.

2. Create a new app and register it

Create a new app named 'user_model' for practice purposes.

Command Line - INPUT

```
(venv)customer_user % | python manage.py startapp user_model
```

Add the app in settings.py.

```
config/settings.py
```

```
INSTALLED_APPS = [  
:  
'user_model',  
]
```

3. Create a custom user model

Create the CustomUser model in models.py. Add the 'mobile' field to compare with the previous practice later.

```
user_model/models.py
```

```
from django.db import models  
  
from django.contrib.auth.models import AbstractUser  
  
class CustomUser(AbstractUser):  
  
    class Meta(AbstractUser.Meta):  
  
        db_table = 'custom_user'  
  
        mobile = models.CharField(max_length=15, blank=True)
```

4. Register the custom user model in settings.py

This is an important step. The Django built-in user model is currently used for the Django user authentication system (e.g., logging in to the Django admin page). By adding this setting, the custom model will be used for the authentication system.

```
config/settings.py
```

```
AUTH_USER_MODEL = 'user_model.CustomUser'
```

5. Edit admin.py

To add the CustomUser model to the admin page, edit the admin.py file under the user_model directory.

```
user_model/admin.py
```

```
from django.contrib import admin  
  
from .models import CustomUser  
  
admin.site.register(CustomUser)
```

6. Check the results

Run the makemigrations and migrate commands as we created a new model. As this is a new project, run the createsuperuser command as well. Then, run the runserver command.

Command Line - INPUT

```
(venv)customer_user % | python manage.py makemigrations users  
  
(venv)customer_user % | python manage.py migrate  
  
venv)customer_user % | python manage.py createsuperuser  
  
(venv)customer_user % | python manage.py runserver
```

Go to localhost:8000/admin. You can see that the custom user model is registered under USER_MODEL and that the mobile field has been added.

Note: Custom User Model

According to the Django official documentation, creating a custom user model is highly recommended.

"If you're starting a new project, it's highly recommended to set up a custom user model, even if the default User model is sufficient for you. This model behaves identically to the default user model, but you'll be able to customize it in the future if the need arises. Changing AUTH_USER_MODEL after you've created database tables is significantly more difficult since it affects foreign keys and many-to-many relationships, for example."

[Django documentation reference: Custom User Model](#)

If you are planning to use a custom user model, it is better to avoid applying migrations until you complete the custom user model setup.

Login Required – LoginRequiredMixin

Once you build user authentication functionalities, you may want to control web page access based on users' login status. You can implement this very quickly with LoginRequiredMixin in the class-based view.

LoginRequiredMixin

LoginRequiredMixin is used to check a user's login status and control page access based on the login status. Adding LoginRequiredMixin is very simple, and you just need to inherit LoginRequiredMixin in the class-based view, in which you want to control user access based on the login status.

Two steps to implement:

- Import LoginRequiredMixin from django.contrib.auth.mixins
- Add LoginRequiredMixin as a parent class of the view

You must add LoginRequiredMixin before the generic view (e.g., ListView). If you put LoginRequiredMixin after the generic view, it may not work properly.

Practice

Objective:

Learn how to use LoginRequiredMixin

In this practice, we'll demonstrate how to add user access restrictions for the CRUD pages in the employee learning app.

1. Edit views.py

Edit views.py like in the code below.

```
employee_learning/views.py

:
from django.contrib.auth.mixins import LoginRequiredMixin

:
class CourseList(LoginRequiredMixin, ListView):
:
class CourseDetail(LoginRequiredMixin, DetailView):
:
class CourseCreate(LoginRequiredMixin, CreateView):
:
class CourseUpdate(LoginRequiredMixin, UpdateView):
:
class CourseDelete(LoginRequiredMixin, DeleteView):
:
```

2. Check the results

Execute the runserver command and go to <http://localhost:8000/employee-learning/course-list/>. You can see that the page is redirected to the Sign In page. If you are logged in, log out first and click the link to the List page.

Note: Login Required Decorator

The `LoginRequiredMixin` approach works only for the class-based view. If you want to add the same functionality for the function-based view, add `@login_required` before the view. The `login_required` module is a decorator. A decorator in Python can add new functionality to an existing object (e.g., class) without modifying the object itself.

The code below is an implementation example. To use the decorator, you need to import it first.

`views.py`

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

@login_required
def detailview(request, pk):
    :
```

User Login Status Icon on Navigation Bar

From the user experience point of view, you may want to show the user's login status by displaying a user icon and user name when the user logs in. This lesson will explain how to show the user icon and user name on the navigation top bar.

There are four design points to implement this.

1. Display a username
2. Adjust the position with Bootstrap
3. Add a user icon
4. Add logic to display only when a user logs in

1. Display a username

To display a username, use two tags – { % load account %} and { % user_display user %}. Place { % user_display user %} where you want to display a username. And, add { % load account %} before { % user_display user %}.

2. Adjust the position with Bootstrap

You need to create another tag with the navbar-nav class to display the user name on the top right of the corner. Add ms-auto (margin start auto) in the tag.

3. Add a user icon

First, you need to place a user icon in the images directory under the static directory. Then, add `src="{% static 'images/user.svg'%}"` and `width="30px"` in the image tag. This is a still static icon. To customize a user icon, please check another course.

4. Add logic to display only when a user logs in

As we don't want to show the user icon when the user isn't logged in, add a condition using the `{% if %}` tag.

This is the code example of the updated navbar.html.

templates/base/navbar.html

```
<div class="collapse navbar-collapse" id="navbarNav">  
<ul class="navbar-nav">  
  
<li class="nav-item"><a class="nav-link" href="{% url  
'course_list'%}">List</a></li>  
  
:  
  
<li class="nav-item"><a class="nav-link" href="{% url 'account_signup'  
'%}">Sign Up</a></li>  
  
{% endif %}  
  
</ul>  
  
{% load account %}  
  
{% if user.is_authenticated %}  
  
<ul class="navbar-nav ms-auto">  
  
<li class="nav-item">
```

```
<a class="nav-link" href="#">![User icon](% static 'images/user.svg')% user_display user %></a>
</li>
</ul>
{%
else %
{%
endif %
</div>
```

Note: ms (margin start) and me (margin end)

To adjust horizontal margins, ml-auto or mr-auto were used before. Bootstrap 5 introduced new class names. ml-* (margin left) and mr-* (margin right) have changed to ms-* (margin start) and me-* (margin end) in Bootstrap 5.

Chapter 6

Deploy Django App

This chapter covers how to deploy your Django app in the production environment. We'll also explain how to manage several software tools and services through the deployment process, such as AWS Lightsail, GitHub, PostgreSQL, Nginx, GCP (Google Cloud Platform), and SendGrid.

The following topics are covered in this chapter.

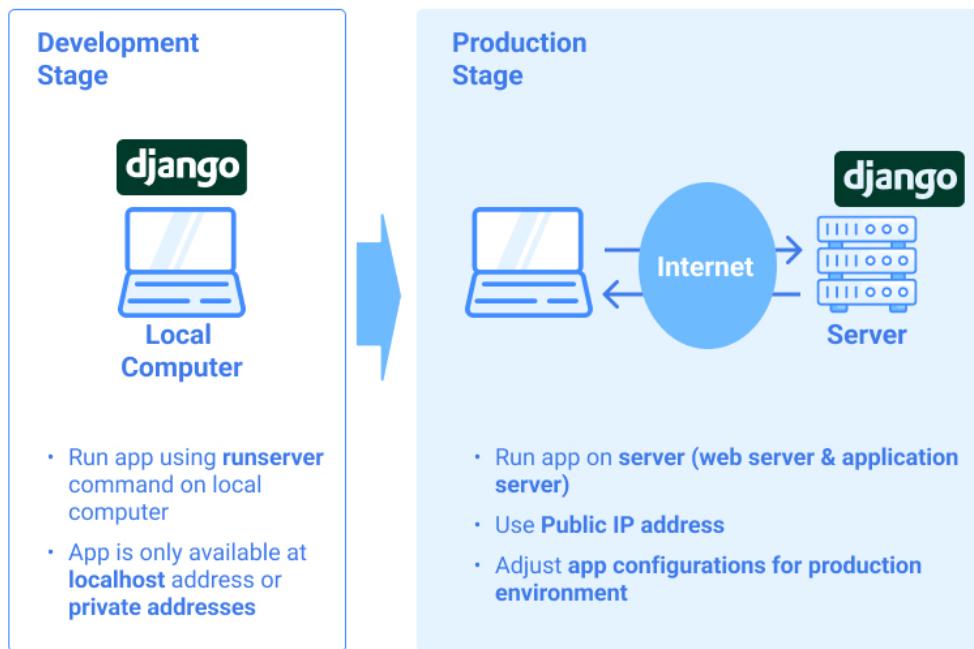
Topics

1. Overview of Django App Deployment (1)
2. Overview of Django App Deployment (2)
3. Key Steps of Django App Deployment
4. Hosting Service Initial Settings (1) – AWS Lightsail setup
5. Hosting Service Initial Settings (2) – SSH Remote Connection
6. Manage Local Computer and Remote Server Simultaneously
7. Tips of Managing Local Development and Remote Production Environment
8. Hosting Service Initial Settings (3) – Clone Project Directory with GitHub
9. Production Database Setup
10. Django App Production Configuration (1) – Settings.py for Development and Production
11. Django App Production Configuration (2) – Production Settings

12. Django App Production Configuration (3) – django-environ and .env file
13. Static File Settings
14. Django and Dependency Installation on Production Server
15. Web Server and Application Server in Django
16. Application Server Setup – Gunicorn
17. Web Server Setup – Nginx
18. Domain Setup
19. SSL Setup – Certbot
20. Email Setting – SendGrid
21. Social Login for Production
22. Manage Local Development and Remote Production Environment – Practice

Overview of Django App Deployment

(1)

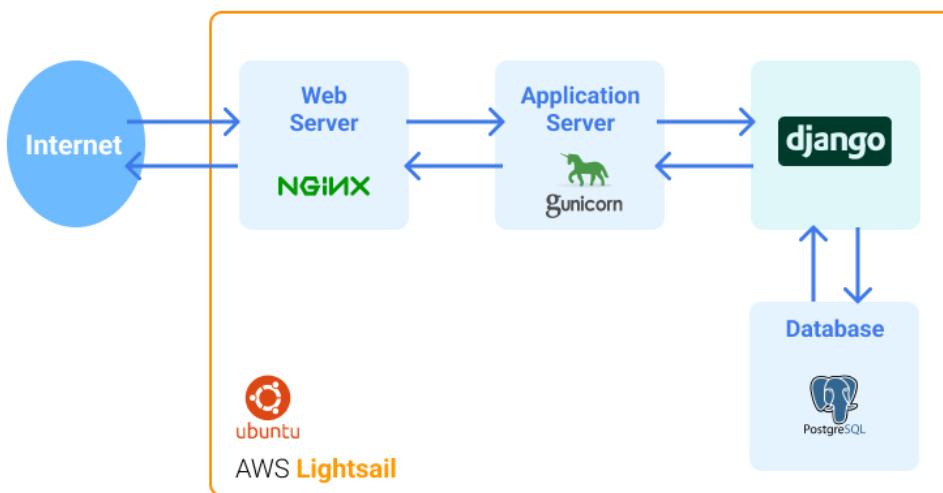


Currently, your app is available only on your computer through the *localhost* address or a **Private IP** address of your local network. Thanks to the `runserver` command, running a Django app is very easy in the local **development environment**.

To make your app accessible to anyone, you need to deploy your Django app on a server. You can use your computer as a server, but usually, we use commercial services, such as the **public cloud**, **VPS** (Virtual Private Server), or a rental server. You also need to use a **Public IP address** so that the app can be accessed through the Internet. As the app will be exposed publicly, you need to adjust app configurations for the **production environment**.

If you haven't done app deployment before, you may feel that the deployment processes are very complex, but if you go through the process once, you'll understand that you can manage it. As there are a lot of potential pitfalls during the Django app deployment process, we'll explain it step-by-step and in as much detail as possible.

Overview of Django App Deployment (2)



In app deployment, there are many choices you need to make. For example:

- Which hosting service are you going to use?
- On which OS do you want to deploy your app?
- Which database are you going to use for production?
- Which combination of the web server and application server are you going to configure?

In this chapter, we'll explain the app deployment processes with specific case examples addressing the questions above.

Hosting Service

The first choice you need to make is which hosting service you want to use. There are different types of hosting services and providers.

For hosting services, the traditional approach was using a **rental server**; however, it is getting less popular now. **Cloud services** or **VPS (Virtual Private Server)** have recently become more popular as you don't need to worry about technical infrastructure management.

There are further different types of cloud services. For example, **PaaS (Platform as a Service)** like **Vercel**, **Firebase**, or **PythonAnywhere** provides easy-to-use services. **IaaS (Infrastructure as a Service)** like **AWS**, **Azure**, or **GCP (Google Cloud Platform)** provides more customizable services. Usually, cloud services use a **pay-per-use** billing model.

VPS is a service similar to IaaS, but its billing scheme is usually fixed price.

In this chapter, we'll use **AWS Lightsail**, which is a VPS service provided by AWS. It offers a three-month free trial.

OS

You need an **OS (Operating System)** to run a Django application. **Linux OS** is the most popular OS for web servers. Linux OS has many **distributions**, such as **Ubuntu**, **CentOS**, **Debian**, and **Fedora**. In this chapter, we use **Ubuntu OS (20.04 LTS)**. You need basic knowledge of Linux OS to deploy your Django app on Linux OS. If you want to learn it, please check our **Linux OS Introduction** course.

[**Linus OS Introduction**](#)

Database

The default database for Django is **SQLite**, which is free open-source database software. It is a lightweight and serverless database, mainly used in small to medium-scale applications. It can be used at the production stage, but **MySQL** or **PostgreSQL** are more popular for production use. You can also use cloud database services. In this chapter, we use **PostgreSQL**.

Web Server and Application Server

Two types of server functionalities are required for dynamic web applications – **web server** and **application server**. The key difference between those two servers is that the web server manages **static content** while the application server handles **dynamic content**.

Depending on the server software, the application server functionality can be part of web server software. For example, for **Apache**, which is popular web server software, the application server functionality is covered by an extended module. On the other hand, another popular web server, **Nginx**, focuses more on web server functionalities.

In this chapter, we use **Nginx** for a web server and **Gunicorn** for an application server.

For the application server functionality, Django supports two standard interfaces: **WSGI (Web Server Gateway Interface)** and **ASGI(Asynchronous Server Gateway Interface)**. Gunicorn is a WSGI server.

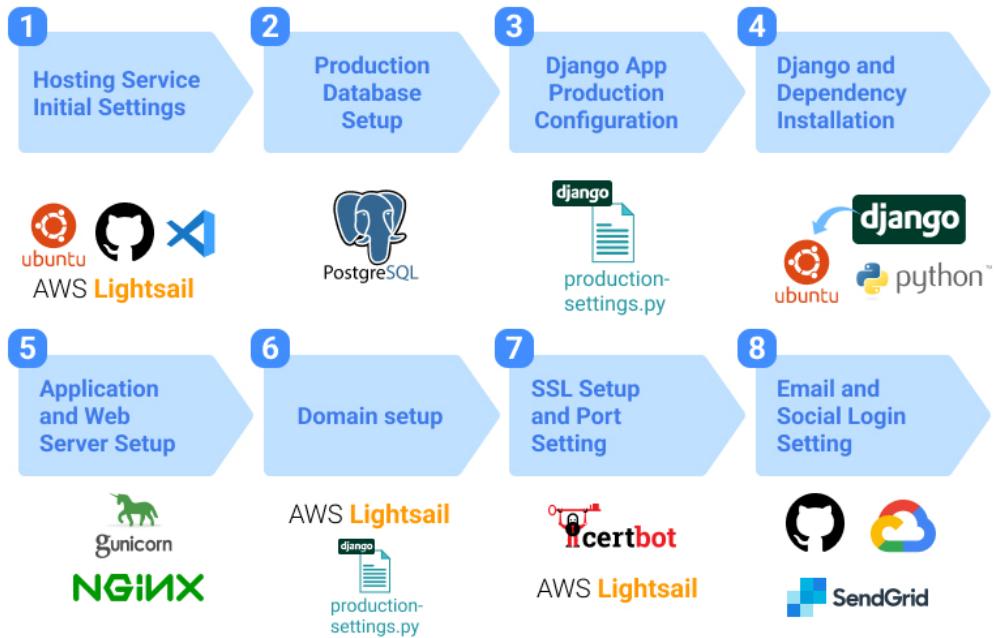
Summary

Here is the summary of software and service choices for our Django app deployment in this chapter.

- Hosting Service: **AWS Lightsail**
- OS: **Ubuntu (20.04 LTS)**
- Database: **PostgreSQL**
- Application Server: **Gunicorn**
- Web Server: **Nginx**

For the virtualization approach, we continue to use **Python venv**. Using **Docker** has recently become an alternative and more popular approach. We'll explain how to deploy a Django app using Dockar in another course.

Key Steps of Django App Deployment



There are eight key steps to deploy a Django app from scratch. If you are handling app deployment for the first time, it may take time to go through all the processes, but once you have familiarized yourself with them, you can manage the deployment a lot faster. As there are many potential pitfalls in the deployment process, we try to explain it in as much detail as possible in this chapter. Here is an overview of the eight key steps in the Django app development. You don't need to understand everything now. We'll go through them one by one in this chapter.

1. Hosting Service Initial Settings

- 2. Production Database Setup – PostgreSQL setup**
- 3. Django App Production Configuration – Edit settings.py**
- 4. Django and Dependency Installation**
- 5. Application and Web Server Setup**
- 6. Domain Setup**
- 7. SSL Setup and Port Setting – Certbot setup**
- 8. Social Login and Email Settings**

1. Hosting Service Initial Settings

Once you decide on a hosting service (e.g., **AWS Lightsail**) for your Django app, you need to sign up for the hosting service and set up a server (**instance**) with a specific OS (e.g., **Ubuntu**).

As the default IP address is usually a dynamic IP address, you need to get a **static IP address** and attach it to the instance.

You also need to connect your local computer to the server using **SSH** to control the server from your local computer.

You can use **VS Code** to edit code remotely in the GUI environment. To transfer your code from the local computer to the server, **GitHub** (a git repository platform) is often used.

At this stage, you also need to confirm that a correct **Python** version has been installed on the server.

2. Production Database Setup – PostgreSQL setup

To change a database from the default database **SQLite**, you need to install database software that you want to use in the production stage (e.g., **PostgreSQL**).

You also need an initial setup for the database.

3. Django App Production Configuration – Edit `settings.py`

In the production stage, you need to adjust several development configurations.

You'll need to continue using two settings – one for development and one for production.

One of the approaches to handle two or multiple settings is to create **multiple settings files for switching configurations** between the development environment and production environment.

To manage highly confidential information, such as passwords or secret keys, you need to make an `.envfile` and save the information separately from the `settings.py` file.

4. Django and Dependency Installation

Django software is usually not preinstalled on a server, so you need to install it for deployment.

To make sure that the **computing environment** is the same as the one on your local computer, you need to install all required software under the **virtual environment** (in the status where `venv` is activated).

You also need to install the required libraries for your app. For example, **gunicorn** for an application server and **psycopg2-binary** for a PostgreSQL adapter.

5. Application and Web Server Setup

Application Server Setup (Gunicorn)

As Gunicorn (an application server) runs in a different process from Django on Linux OS, you need to make additional settings to run it.

- Create two unit files (**gunicorn.socket** and **gunicorn.service**)
- Enable the unit files using the `systemctl` command.

With these settings, Gunicorn will automatically start and run as a background job when the server boots.

Web Server Setup (Nginx setup)

If **Nginx** is not preinstalled, you need to install it first.

To customize the Nginx configuration to your development environment and the Django app, you need to create a configuration file under `/etc/nginx/sites-available/`.

As Nginx handles static files directly, so you only need to define the static file path in the configuration file.

The static file path should be the same as the one defined by `STATIC_ROOT` in `settings.py`.

To enable the configuration, you need to create a symbolic link of the configuration file under `/etc/nginx/sites-enabled/`.

6. Domain Setup

You may want to make your web application accessible through domain name (not through IP address only). To convert an IP address to domain name, you need to register a

domain name and add a DNS record in the DNS server. As Django limits host names that can access the application, you need to edit `ALLOWED_HOSTS` in the setting file for production along with the Nginx configuration adjustment.

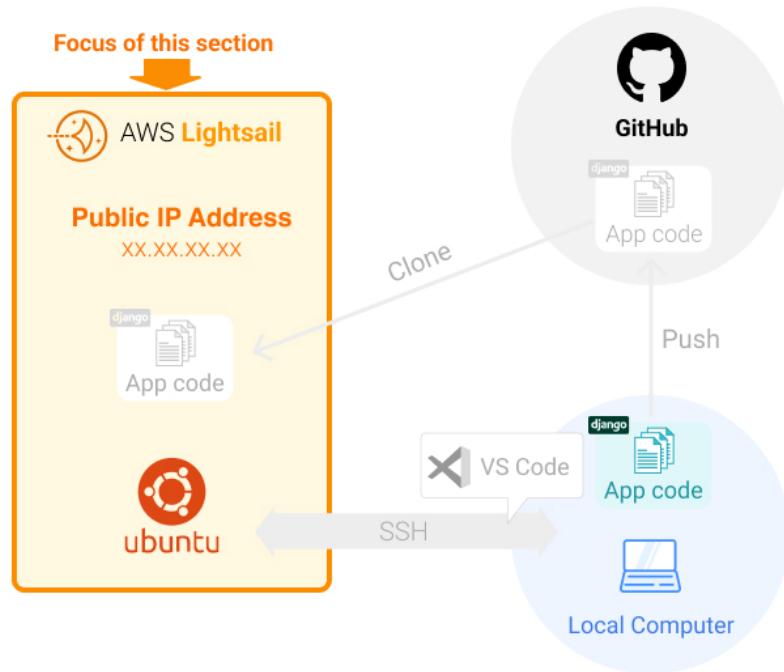
7. SSL Setup and Port Setting – Certbot setup

To build a more secured communication using **HTTPS**(Hypertext Transfer Protocol Secure), you need to set up **SSL** (Secure Sockets Layer). **Certbot** is a useful tool for setting up SSL. It is a free open-source software tool that automatically provides a **Let's Encrypt certificate** used for SSL communication. As ports for HTTP and HTTPS are different, you need to change **port settings** to open in the **firewall** setting.

8. Social Login and Email Settings

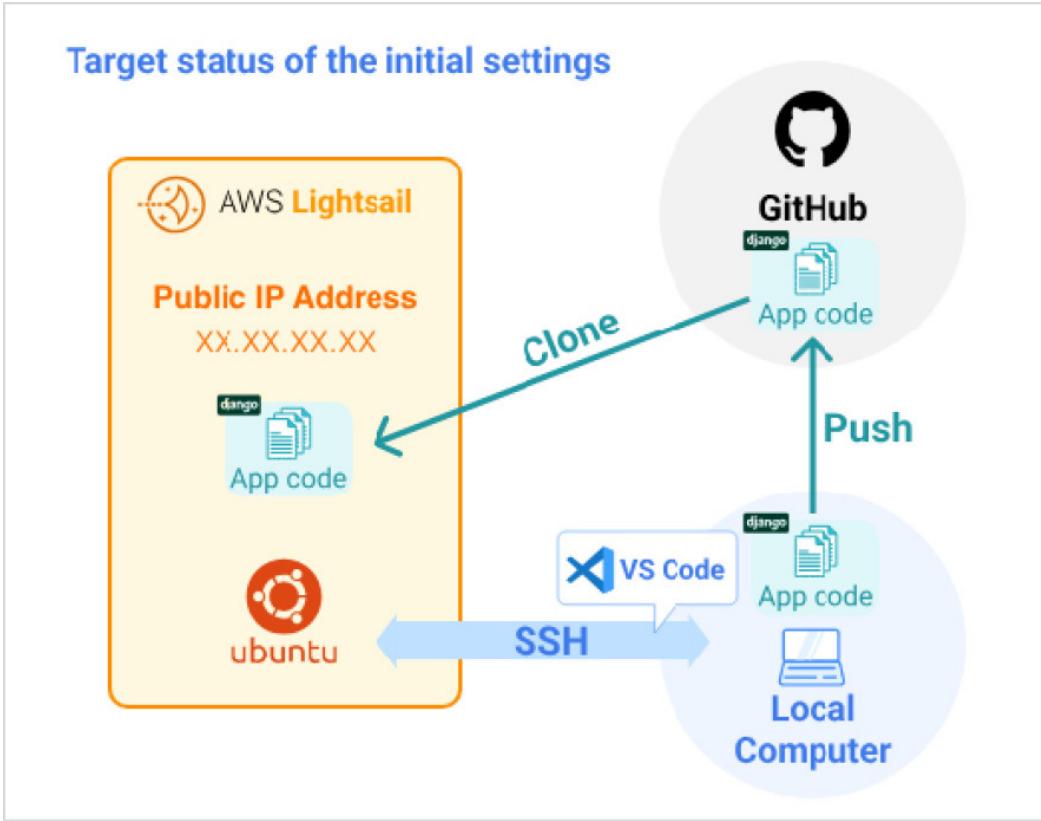
Lastly, you need to adjust social login settings as you start to use a different host address. You need to register it in each social login provider's platform and Django admin. Also, for the production, there are some platforms that can send machine-generated emails. Using one of the email platforms is preferable from security and reliability point of view. **SendGrid** is one of the popular choices.

Hosting Service Initial Settings (1) – AWS Lightsail setup



Summary of the initial settings

The illustration below shows the initial settings' target status to place the app code on the webserver. If you are managing this step for the first time, it may take longer than you expected, as this step requires a lot of configurations for different platforms. Some processes are not mandatory, but they give you operational efficiency later on.



We split this step into three sections. This lesson covers the first section.

1. AWS Lightsail setup (this section)

- Prepare AWS Lightsail and create an Ubuntu (20.04 LTS) instance
- Obtain a static IP address and attach it to the instance

2. SSH remote connection between the local computer and the Ubuntu instance

- Establish an SSH connection between the local computer and the instance
- Enable the remote access using VS Code

3. Clone the project directory to the instance using GitHub

- Push the project directory from the local computer to the GitHub repository
- Establish a remote connection between the GitHub repository and the Ubuntu instance (register a developer key)
- Clone the project directory to the Ubuntu instance

Prepare AWS Lightsail and create an Ubuntu (20.04 LTS) instance

We have detailed explanations on how to set up **AWS Lightsail** in our '**Linux OS Introduction**' course. Click the button below to open the related page in a different tab.

[Linux OS Introduction 'Setting Up Linux Environment on AWS'](#)

Obtain a static IP address and attach it to the instance

Once you create a **Ubuntu** instance, you need to obtain a **static IP address** and attach the address to the instance.

[**Go to the Networking section in your AWS Lightsail console to obtain a static IP address. Click on the Create static IP button.**](#)

The screenshot shows the AWS Lightsail Networking section. At the top, there's a greeting "Good afternoon!" and a search bar labeled "Filter by name, location, tag, or type". Below the header, there are tabs for Instances, Containers, Databases, **Networking**, Storage, Domains & DNS, and Snapshots. The Networking tab is selected. A sub-section titled "Connect your project!" explains that networking resources allow specifying how users and outside services connect to your Lightsail resources, mentioning routing, internet traffic, redundancy, and content delivery. Two main options are presented: "Static IP" and "Distribution". The "Static IP" section features an icon of a wrench and screwdriver, a brief description, a "Learn more about static IPs" link, and a prominent orange "Create static IP" button. The "Distribution" section features an icon of a map pin with arrows, a brief description, a "Learn more about distributions" link, and an orange "Create distribution" button. An orange rectangular box highlights the "Create static IP" button in the Static IP section.

Select your instance to attach the static IP and click the Create button.

Attach to an instance

Attaching a static IP replaces that instance's dynamic IP address.

Static IP addresses can only be attached to instances in the same region.

Select an instance... ▾

Ubuntu-1

Identify your static IP

Your Lightsail resources must have unique names.

StaticIp-1

Static IP addresses are free only while attached to an instance.
You can manage five at no additional cost.

Create

You can confirm your public IP address attached to the Ubuntu OS instance.

StaticIp-1

Static IP, Attached

Virginia, all zones (us-east-1)

Static IP: 44.212.196.197

Details Domains

Public static IP address

This static IP is available for public connection worldwide.

44.212.196.197

Attach to an instance

Attaching a static IP replaces that instance's dynamic IP address.

Ubuntu-1

2 GB RAM, 1 vCPU, 60 GB SSD

Ubuntu

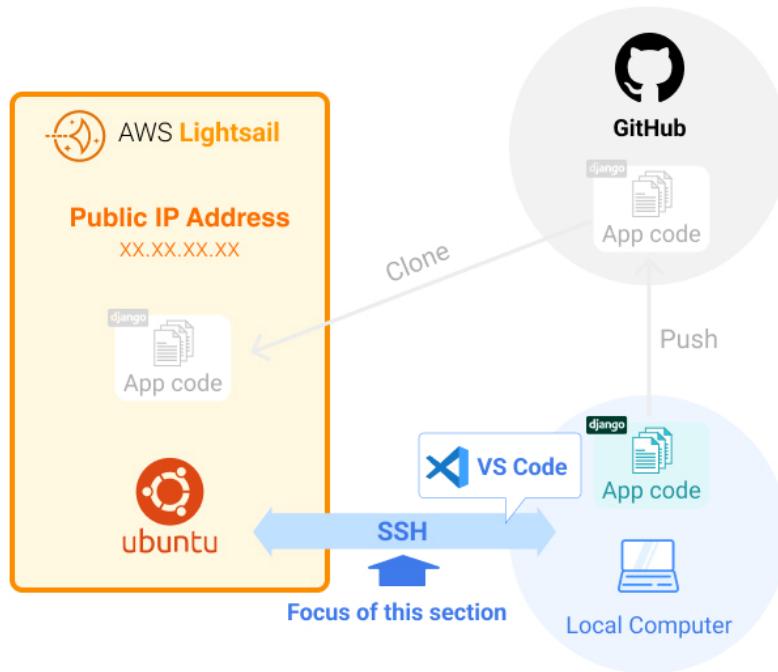
Detach X

Using a static IP address, you can continue to use the same IP address even when you delete the current instance. When you create a new instance, you just need to re-attach the static IP address to the new instance.

This static public IP address will be used for your **SSH** connection settings in the next section.

If you are not using a static IP address, the IP address will change every time you switch to a new instance. Using a static IP address gives you better operational efficiency as you don't need to re-set the SSH configuration.

Hosting Service Initial Settings (2) – SSH Remote Connection



Summary of the initial settings

We split this step into three sections. This lesson covers the second section.

1. AWS Lightsail setup

- Prepare AWS Lightsail and create an Ubuntu (20.04 LTS) instance
- Obtain a static IP address and attach it to the instance

2. SSH remote connection between the local computer and the Ubuntu instance (this section)

- Establish an SSH connection between the local computer and the Ubuntu instance
- Enable the remote access using VS Code

3. Clone the project directory to the instance using GitHub

- Push the project directory from the local computer to the GitHub repository
- Establish a remote connection between the GitHub repository and the Ubuntu instance (register a developer key)
- Clone the project directory to the Ubuntu instance

Establish an SSH connection between the local computer and the Ubuntu instance

We also cover how to establish an SSH remote connection in the '**Linux OS Introduction**' course. Click on the button below to open the related page in a different tab.

[Linux OS Introduction 'SSH Remote Login \(1\) – Use Key Pair Generated by Server'](#)

Also, it is better to use an SSH configuration file for quicker remote access setup. Refer to this topic in the '**Linux OS Introduction**' course.

[Linux OS Introduction 'SSH Config File'](#)

Enable remote access using VS Code

By now, you should be able to access your Ubuntu OS from your terminal; however, you cannot see the Linux directory tree and files. Using **VS Code** extensions, you can access the Ubuntu OS instance through VS Code, and you'll be able to see the directory tree and files in GUI (Graphical User Interface).

We also covered how to establish an SSH remote login with VS Code in the '**Linux OS Introduction**' course. Click on the button below to open the related page in a different tab.

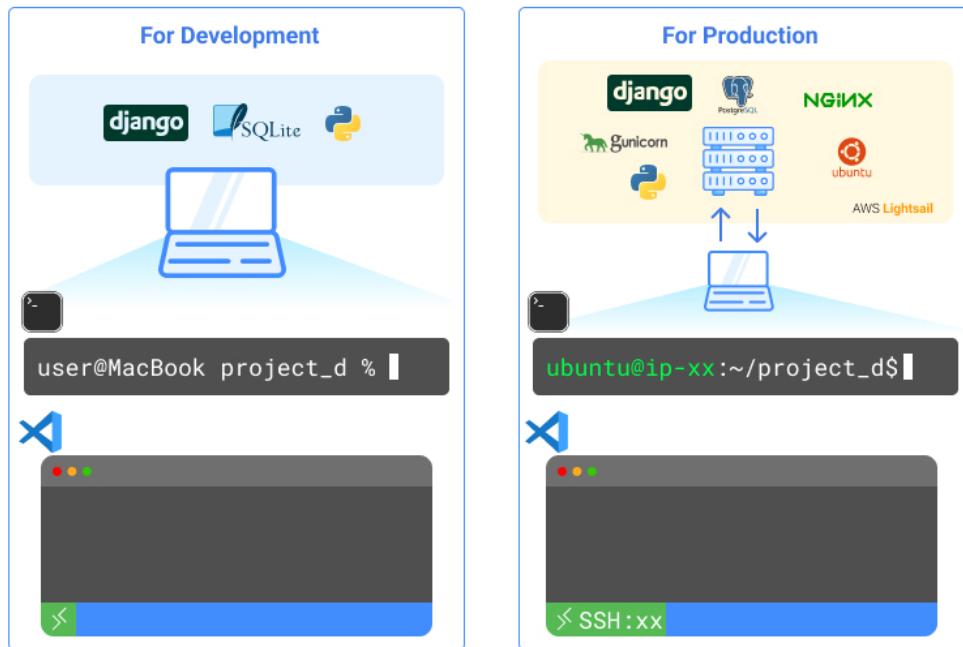
[**Linux OS Introduction 'SSH Remote Login with Visual Studio Code'**](#)

When you successfully establish the remote login, your terminal in VS Code will be like the one below.

Command Line - INPUT

```
ubuntu@ip-172-26-8-65:~$
```

Manage Local Computer and Remote Server Simultaneously



Now you can access the server (**Ubuntu Instance**) remotely using **VS Code** and terminal. This means you are managing two machines on your local computer. One is your local computer itself, and the other is the remote server.

In the app deployment process, you need to be clear about which computer you are handling.

You can open two VS Code windows with a terminal inside to manage the two computer resources. You can check which computer resources (remote vs. local) you are accessing from the two points.

1. From the terminal window

This is for the remote server. The command prompt starts with the remote server's **user name** and the **private IP address**. Usually, they are colored.

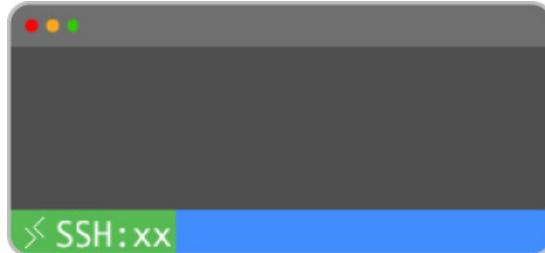
```
ubuntu@ip-xx:~/project_d$ |
```

This is for the local computer. The command prompt starts with **your local computer user name** and **machine name**.

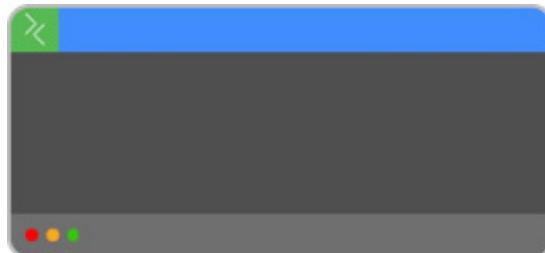
```
user@MacBook project_d % |
```

2. From the VS Code bottom status bar

This is an illustration of the remote server. Information about the remote login name (SSH name) is shown at the bottom left.



This is for the local computer. There is **no information** about SSH.



Tips for Managing Local Development and Remote Production Environment

	Control Point	Examples
Do not share irrelevant files	.gitignore	.env file, virtual environment directory, local database file (SQLite file) etc.
Do not install irrelevant dependencies	requirements.txt	For production gunicorn, PostgreSQL adapter, etc.
Do not set irrelevant configurations	settings.py	For production DEBUG=False, ALLOWED_HOST=[Server Public IP address], etc.

As the requirements (e.g., security, specs, scalability, etc.) of the **local development environment** and **the production environment** on a server are different, you need to manage those differences properly and carefully.

There are three key points in managing the differences in the deployment process.

1. Do not share irrelevant files (.gitignore)
2. Do not install irrelevant dependencies (requirements.txt)
3. Do not set irrelevant configurations (settings.py)

1. Do not share irrelevant files (`.gitignore`)

As some directories and files are not relevant to the production environment, you should not transfer the directories and files that are irrelevant to the production server. For example, the `.env` file is highly confidential and should not be shared. (we'll explain about `.env` later).

Also, the **virtual environment** directory (for example, `d_env` in our course case study) depends on the machine you have installed. You need to create another virtual environment directory on your server. Thus, you don't want to transfer the directory to the server.

To avoid sharing irrelevant files and directories when you are using **Git**, you can use the `.gitignore` file. We'll explain how to create `.gitignore` for Django deployment on the next page.

2. Do not install irrelevant dependencies (`requirements.txt`)

In principle, you want to create the same environment on both the local machine and the server; however, there should be some adjustments. As explained, you may want to switch your database software or need additional libraries only for the production environment.

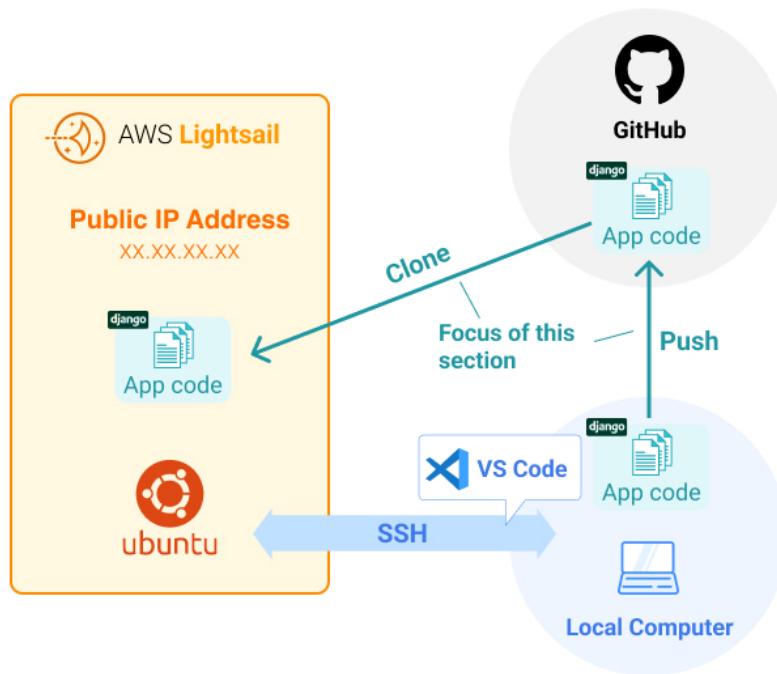
One of the best practices for managing different dependencies is via creating different text files to list requirements for each environment. We'll explain this later in this chapter.

3. Do not set irrelevant configurations (`settings.py`)

You need to adjust several settings in the production environment, especially for security reasons. For example, you want to change the `DEBUG` from `True` to `False`, so that normal users cannot see the detailed error logs when they encounter errors. The details will be explained later in this chapter.

As these points are critical, we'll explain how to manage the three key points in detail in this chapter and summarize the points at the end of this chapter.

Hosting Service Initial Settings (3) – Clone Project Directory with GitHub



Summary of the initial settings

We split this step into three sections. This lesson covers the last section.

1. AWS Lightsail setup

- Prepare AWS Lightsail and create an Ubuntu (20.04 LTS) instance
- Obtain a static IP address and attach it to the instance

2. SSH remote connection between the local computer and the Ubuntu instance

- Establish an SSH connection between the local computer and the instance
- Enable the remote access using VS Code

3. Clone the project directory to the instance using GitHub (this section)

- Push the project directory from the local computer to the GitHub repository
- Establish a remote connection between the GitHub repository and the Ubuntu instance (register a developer key)
- Clone the project directory to the Ubuntu instance

Push the project directory from the local computer to the GitHub repository

To transfer your code from the local computer to the server, GitHub (a git repository platform) is often used. If you are not familiar with Git and GitHub, we recommend that you check our '**Git and GitHub Introduction**' course.

[Git and GitHub Introduction](#)

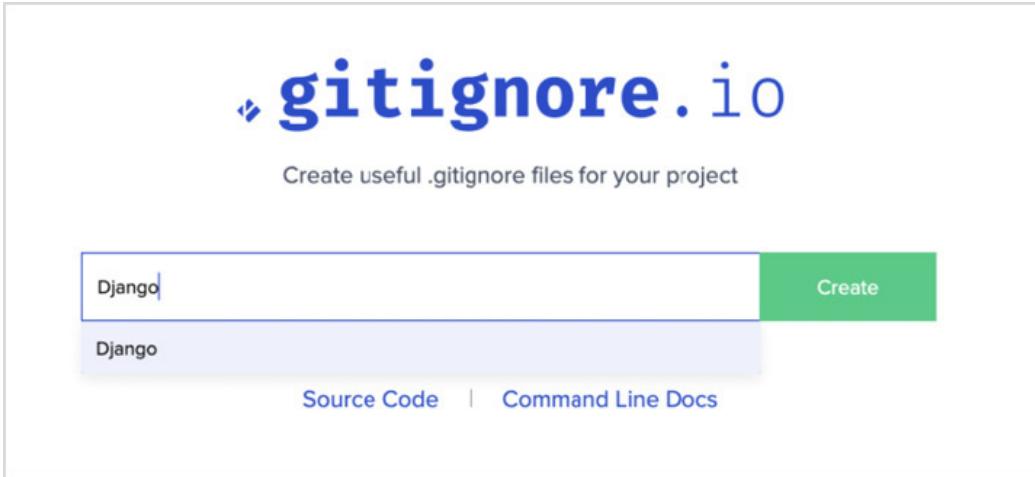
As the process of establishing Git and GitHub requires a lot of explanations. Here, we will mainly cover key points for the Django project code transfer process.

.gitignore

Before you push (send) your project documents to the GitHub repository, you need to create the `.gitignore` file. There are files or directories you don't want to push to the repository.

You can use [gitignore.io](https://www.gitignore.io) to check typical files or directories that should be listed in the `.gitignore` file for Django app.

Go to the website, type '*Django*', and click the **Create** button.



You'll see a list of files or directories like in the example below.

```
# Created by https://www.toptal.com/developers/gitignore/api/django
# Edit at https://www.toptal.com/developers/gitignore?templates=django

### Django ###
*.log
*.pot
*.pyc
__pycache__/
local_settings.py
db.sqlite3
db.sqlite3-journal
media

# If your build process includes running collectstatic, then you probably
# want to add staticfiles/ to your .gitignore. Update and uncomment the following line accordingly:
# <django-project-name>/staticfiles/

### Django.Python Stack ###
# Byte-compiled / optimized / DLL files
*.py[cod]
*$py.class
```

Create the `.gitignore` file directly under the project directory, and copy the list from gitignore.io.

For the virtual environment (`venv`) directory, add **your virtual environment directory name with / at the end**. In our case example, add `d_env/` to the list.

Push the code to a GitHub repository

After creating a GitHub repository (e.g., *project_d*), commit the latest code and push the code to the repository.

Establish a remote connection between the GitHub repository and the Ubuntu instance (register a developer key)

Here are the key steps to register a **developer key**.

1. Create an ssh key pair from the Ubuntu instance

To generate a key pair, run the command below with your email address.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | ssh-keygen -t ed25519 -c your_email@example.com
```

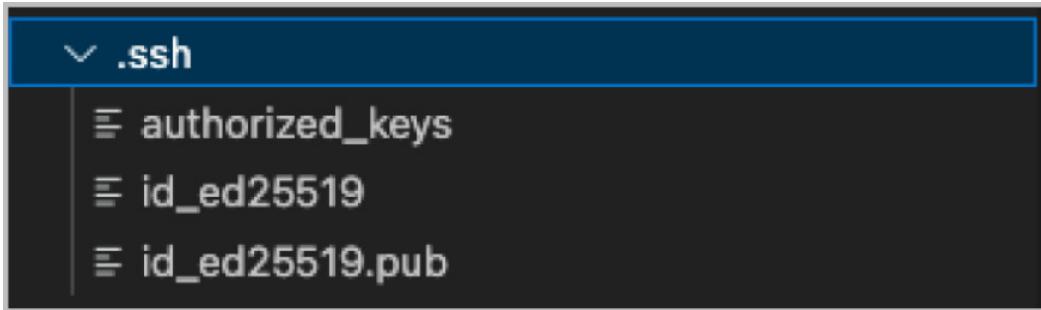
You'll be asked to type a passphrase, but you don't need to type it unless you want to increase the security level. If you set a passphrase, you'll need to type it every time you access your GitHub repository later.

Command Line - INTERACTIVE

```
Generating public/private ed25519 key pair.  
Enter file in which to save the key  
(/home/ubuntu/.ssh/id_ed25519):  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:
```

2. Copy the public key

The public key is available under the .ssh directory located under the home directory (i.e., the *ubuntu* directory). Copy the key and go to the next step.

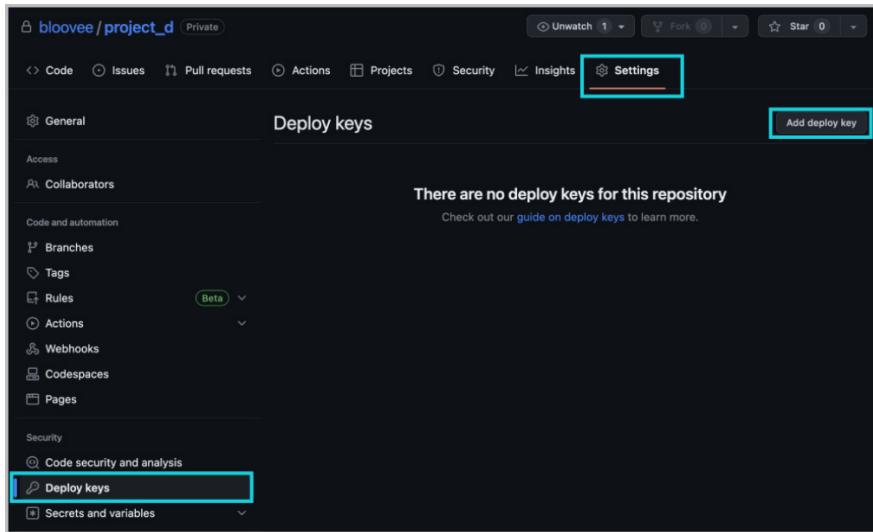


.ssh/id_ed25519.pub

```
ssh-ed25519 xxxxxxxxx your_email@example.com
```

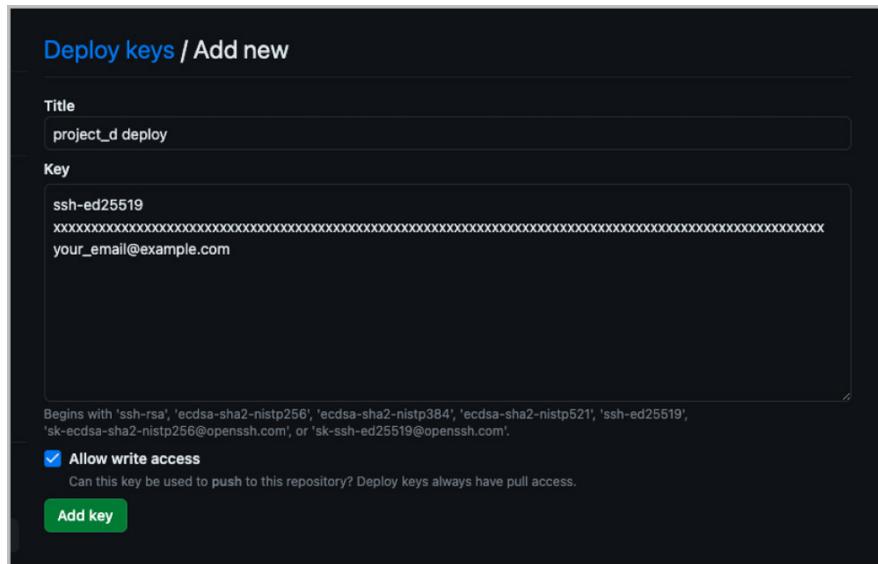
3. Add the public key to the GitHub repository

Go to your GitHub repository for this project. Under the **Settings** tab, click on the **Deploy keys** on the left sidebar. Press the **Add deploy key** button.



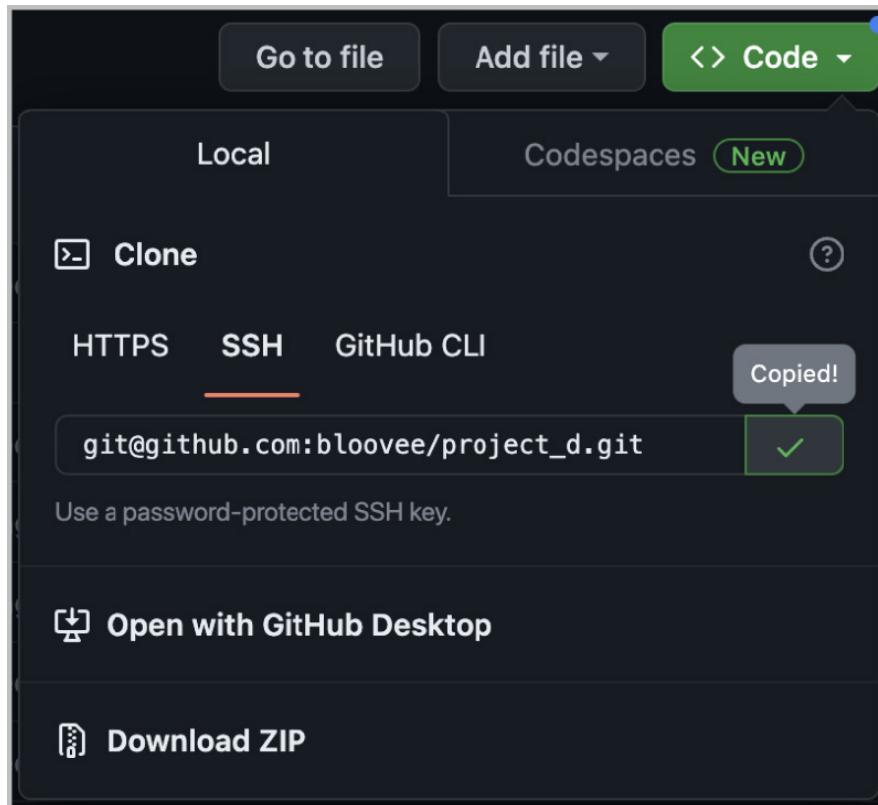
Add the **Title** and paste the copied key. Check on **Allow write access**. This enables you to push the code from the Ubuntu instance to the GitHub repository.

Finally, press the **Add key** button to complete the process.



Clone the project code to the Ubuntu instance

Go to the repository and open the **<>Code** dropdown. Copy a URL for SSH.



Run the git clone command **from the home directory** with the copied URL.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | cd  
ubuntu@ip-xx:~$ | git clone [copied url]
```

You'll get the project directory in the Ubuntu instance.

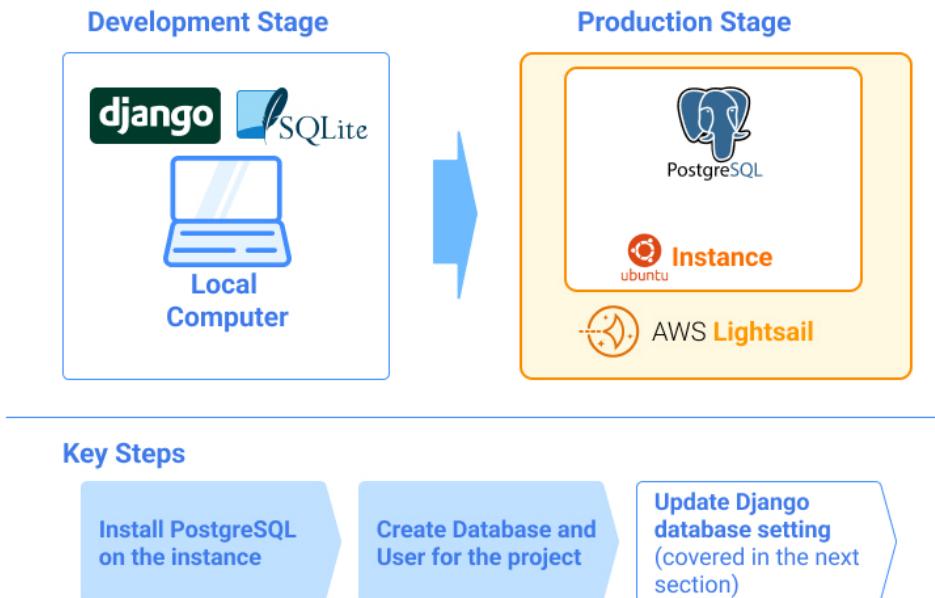
Command Line - INPUT

```
ubuntu@ip-xx:~$ | ls
```

Command Line - RESPONSE

```
project_d
```

Production Database Setup



To change a database from the default database **SQLite**, you need to install database software that you want to use in the production stage (e.g., **PostgreSQL**). You also need to do the initial setup for the database.

Note: Most actions from this lesson must be done on the Ubuntu instance. Make sure that you are connected to the remote server via SSH.

1. Install PostgreSQL on the instance

Based on the PosgtreSQL official documentation, you can install PostgreSQL in the following steps.

1. Create the file repository configuration

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo sh -c 'echo "deb  
http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg  
main" > /etc/apt/sources.list.d/pgdg.list'
```

2. Import the repository signing key

Command Line - INPUT

```
ubuntu@ip-xx:~$ | wget --quiet -O -  
https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key  
add -
```

3. Update the package lists

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo apt-get update
```

4. Install the latest version of PostgreSQL

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo apt-get -y install postgresql
```

You can check if the PosgreSQL is properly installed and running. First, check the version installed by the `psql --version` command.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | psql --version
```

Command Line - RESPONSE

```
psql (PostgreSQL) 15.2 (Ubuntu 15.2-1.pgdg20.04+1)
```

Using the `systemctl` command, check if PostgreSQL is running as a **demon process**.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | systemctl status postgresql
```

Command Line - RESPONSE

```
● postgresql.service - PostgreSQL RDBMS
```

```
Loaded: loaded (/lib/systemd/system/postgresql.service; enabled;
         vendor preset: enabled)
Active: active (exited) since Thu 2023-04-20 13:13:41 UTC; 2min 20s
         ago
Main PID: 8430 (code=exited, status=0/SUCCESS)
Tasks: 0 (limit: 2372)
Memory: 0B
CGroup: /system.slice/postgresql.service

Apr 20 13:13:41 ip-172-26-0-231 systemd[1]: Starting PostgreSQL
RDBMS...
Apr 20 13:13:41 ip-172-26-0-231 systemd[1]: Finished PostgreSQL
RDBMS.
```

Check the official document for more details about the installation process.

[PostgreSQL documentation reference: Linux downloads \(Ubuntu\)](#)

2. Create a database and user for the project

You need to create a database and user for the project. As the parameters will be used in the Django settings later, copy and save them somewhere.

Start PostgreSQL command prompt

First, you need to go into PostgreSQL using the following command. The prompt changes to `postgres=#`. You can type the [PostgreSQL](#) commands in this mode.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo -u postgres psql
```

Command Line - INTERACTIVE

```
psql (15.2 (Ubuntu 15.2-1.pgdg20.04+1))
Type "help" for help.

postgres=#
```

Create a database for the project

To create a database for the project, run the command below. Don't forget ; at the end.

Command Line - INPUT

```
postgres=# | CREATE DATABASE project_d;
```

When this is completed successfully, the command line returns the message below.

Command Line - RESPONSE

```
CREATE DATABASE
```

Create a user for the project

To create a user for the project, run the command below with the password you want to set.

Command Line - INPUT

```
postgres=# | CREATE USER project_d_user WITH PASSWORD  
'project_d_pass';
```

When this is completed successfully, the command line returns the message below.

Command Line - RESPONSE

```
CREATE ROLE
```

Make sure that you keep the username and password for the settings later.

Add user privileges for the database

You also need to grant all privileges for the database to the newly created user.

Command Line - INPUT

```
postgres=# | GRANT ALL PRIVILEGES ON DATABASE project_d TO  
project_d_user;
```

When this is completed successfully, the command line returns the message below.

Command Line - RESPONSE

```
GRANT
```

Alter the database owner

This setting was not required in the older version of PostgreSQL. Now, you need to add this setting by running the command below.

Command Line - INPUT

```
[user@localhost] $ |ALTER DATABASE project_d OWNER TO  
project_d_user;
```

If you don't set this, you may encounter an error in the database migration process. When this is completed successfully, the command line returns the message below.

Command Line - RESPONSE

```
ALTER DATABASE
```

Exit the prompt

Type `\q` to exit from the PostgreSQL prompt.

Command Line - INPUT

```
postgres=# | \q
```

3. Update the Django database setting

You also need to update the database settings in Django. You need to make sure that the parameters set through the PostgreSQL prompt are the same (i.e., NAME, USER, PASSWORD). Below is an example of the database settings. You don't need to update the `settings.py` file yet, as we'll explain how to create two settings files for development use and production use in the following pages.

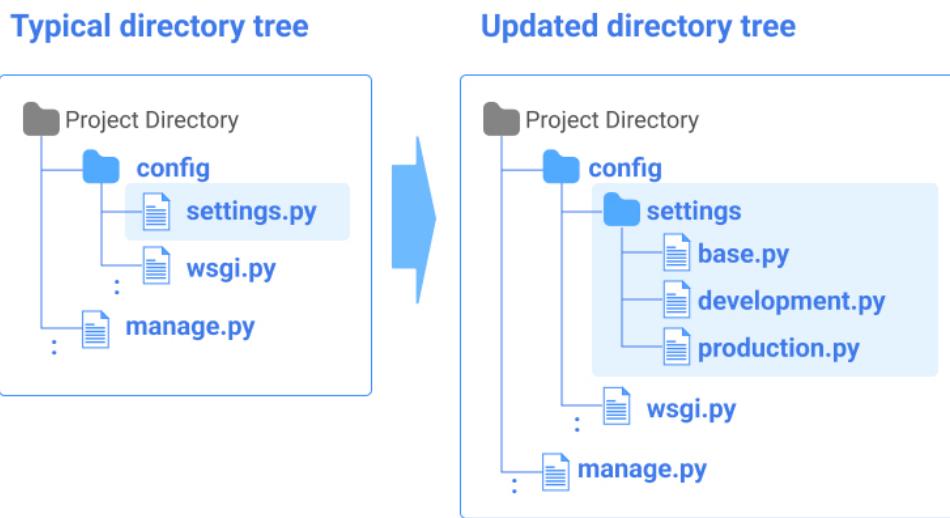
The `yellow` part is a part different from the original `settings.py`, and the blue part is the one you'll need to set for the PostgreSQL. We'll explain how to execute this on the following pages.

config/settings.py

```
:  
  
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'project_d',  
        'USER': 'project_d_user',
```

```
        'PASSWORD': 'project_d_pass',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Django Production Settings (1) – Settings.py for Development and Production



For the **production environment**, there are several development configurations that need to be adjusted. You'll need to continue using two different settings to manage the different environments.

One of the approaches to handling two or multiple settings is creating **multiple settings files to switch configurations** between the local environment and the production environment.

To manage highly confidential information, such as passwords or secret keys, you need to make a `.env` file and save the information separately from the `settings.py` file.

This lesson will explain how to create and manage multiple settings files.

1. Create the settings directory and setting files

There are several use cases for this approach. In our case, we'll create three settings files.

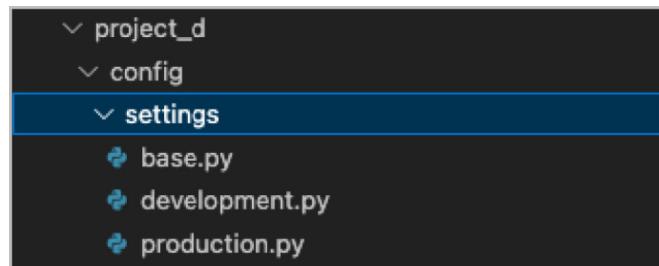
- **base.py**: the original *settings.py* (rename)
- **development.py**: use this for settings for development in the local environment
- **production.py**: use this for settings for production on the server

Here are the key steps.

1. Create the settings directory and save the three files under the directory

Create *development.py* and *production.py*. Rename *settings.py* to *base.py*. Create the *settings* directory. Save the three files under the *settings* directory.

The directory structure will be like the one below.



2. Update base.py

As the directory structure has changed, the `BASE_DIR` path should be updated. Add one more '**parent**' in the path like shown below.

`config/settings/base.py`

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

3. Edit development.py and production.py

Add the following code. The code is the same for both files. With this code, *development.py* and *production.py* can overwrite the settings in *base.py*.

config/settings/development.py

```
from .base import *
```

config/settings/production.py

```
from .base import *
```

2. Update *wsgi.py* and *manage.py*

As the settings file path has changed, you need to adjust the file path to access the new settings files.

manage.py

```
:  
def main():  
    """Run administrative tasks."""  
    os.environ.setdefault('DJANGO_SETTINGS_MODULE',  
        'config.settings.production')  
try:  
:
```

manage.py

```
:  
from django.core.wsgi import get_wsgi_application  
os.environ.setdefault('DJANGO_SETTINGS_MODULE',  
    'config.settings.production')  
:
```

3. Check the result for development settings with the runserver command

To run the app using the local settings, add `--settings config.settings.development` when you execute the `runserver` command.

Command Line - INPUT

```
(d_env) project_d | python manage.py runserver --settings  
config.settings.development
```

With the '`config.settings.local`' setting, you can see that the server is running.

Command Line - RESPONSE

```
System check identified no issues (0 silenced).
April 21, 2023 - 10:27:34
Django version 4.1.7, using settings 'config.settings.local'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

4. Commit and push the code to GitHub from the local computer and pull it from the server

Run the following commands to make the statuses of the local computer and the server the same for the next steps.

From the local

Command Line - INPUT

```
(d_env) project_d | git add .
(d_env) project_d | git commit -m "updated settings structure"
(d_env) project_d | git push origin master
```

From the server

Command Line - INPUT

```
ubuntu@ip-xx:~$ | git pull
```

Command Line - RESPONSE

```
:
config/{settings.py
=> settings/base.py} | 2 ++
config/settings/development.py | 2 ++
config/settings/production.py | 2 ++
config/wsgi.py | 2 ++
manage.py | 2 ++
:
```

Check if the `settings` directory structure is the same as the one on the local computer.

Django Production Settings (2) – Production Settings



production.py

Key settings adjusted for production

DEBUG

ALLOWED_HOSTS

DATABASES

EMAIL_BACKEND and related settings

STATIC_ROOT and STATIC_URL

MEDIA_ROOT and MEDIA_URL

There are several settings you need to update for the production environment. In this section, we'll explain the critical ones. As confidential data should not be directly written in the settings files, you should use the `.env` file, which will be explained in the next section.

Here is the list of the critical settings you want to change for the production environment.

DEBUG

Change to **False**.

With `DEBUG = True`, you can see full tracebacks in your browser when you encounter an error. Exposing the error log information for the production environment is risky as it leaks lots of information about your app: excerpts of your source code, local variables, settings, libraries used, etc.

ALLOWED_HOSTS

Add **public IP address** and **domain** for projection.

This setting limits user access to Django apps. User access will be blocked if the IP address or domain is not listed here.

DATABASES

Change to production database settings. (e.g., PostgreSQL)

Setting parameters differ by the database. As you need to add a password, etc., for the database, you should make sure that you save these settings in the `.env` file.

EMAIL_BACKEND and related settings

Change the email backend to an actual email. In the previous chapter, we explained how to change the email backend to an actual one, even for the development environment. Usually, we use a console for development and an actual email for production. As there is confidential data in email-related settings, you should make sure that you save these settings in the `.env` file for the production environment.

STATIC_ROOT

Add `STATIC_ROOT` to the address where a web server handles static files. We'll explain static file handling for production later in this chapter.

MEDIA_ROOT

Change to the address where a web server handles media files. As we haven't explained media files yet in this course, for now, you just need to remember that the media root directory should be adjusted for production.

These are only some basic settings. For a more comprehensive list, check the Django official documentation.

[Django documentation reference: Deployment checklist](#)

Django Production Settings (3) – django-environ and .env file



As explained on the previous page, we need to manage confidential data relating to the Django app settings. To separate the settings, you can use the **`django-environ`** library.

`django-environ`

`django-environ` allows you to configure the Django application using environment variables obtained from an environment file and provided by the OS.

Basically, using this library, you can separately manage confidential information in another file named `.env`.

Settings written in the `.env` file

Confidential data should be put in the `.env` file. Most confidential data are already included in the settings for production explained on the previous page, except `SECRET_KEY`. `SECRET_KEY` is not specific for production, but it should be confidentially managed and saved under the `.env` file.

Note: `SECRET_KEY`

According to the Django official documentation, `SECRET_KEY` is used to provide cryptographic signing and should be set to a unique, unpredictable value. Running Django with a known `SECRET_KEY` defeats many of Django's security protections and can lead to privilege escalation and remote code execution vulnerabilities.

When you initiated the Django project, the secret key was already generated, and you can use it for production; however, you should not expose it to the public. In practice, you need to make sure that you won't push the information to the GitHub repository.

Key steps to use django-environ

Here are the key steps to use django-environ for our Django app.

1. Install django-environ

Add django-environ in `requirements.txt` and install the package.

`requirements.txt`

```
Django==4.1.7
django-crispy-forms
crispy-bootstrap5
django-allauth
django-environ
```

As we haven't created the virtual environment, actual installation will be done later in this chapter.

2. Edit `production.py`

First, you need to import `environ`, and add initial settings to read the `.env` file.

`config/settings/production.py`

```
from .base import *
import environ

env = environ.Env()
env.read_env('.env')
```

Then, bring the settings for production from the `base.py` file.

And replace confidential data with `env('ENVIRONMENT_VARIABLE')`. If there is a list of parameters, use `env.list('ENVIRONMENT_VARIABLE')`.

Here is an example of `production.py` for our Django app. We don't include email settings and `STATIC_ROOT` here yet. We'll explain them later.

`config/settings/production.py`

```
from .base import *
import environ

env = environ.Env()
env.read_env('.env')

SECRET_KEY = env('SECRET_KEY')

DEBUG = False

ALLOWED_HOSTS = env.list('ALLOWED_HOSTS')

DATABASES = {
    'default': {
        'ENGINE': env('DB_ENGINE'),
        'NAME': env('DB_NAME'),
        'USER': env('DB_USER'),
        'PASSWORD': env('DB_PASSWORD'),
        'HOST': env('DB_HOST'),
        'PORT': env('DB_PORT'),
    }
}
```

And this is the example of the `.env` file. Make sure that the `.env` file is created directly under the project directory.

`.env`

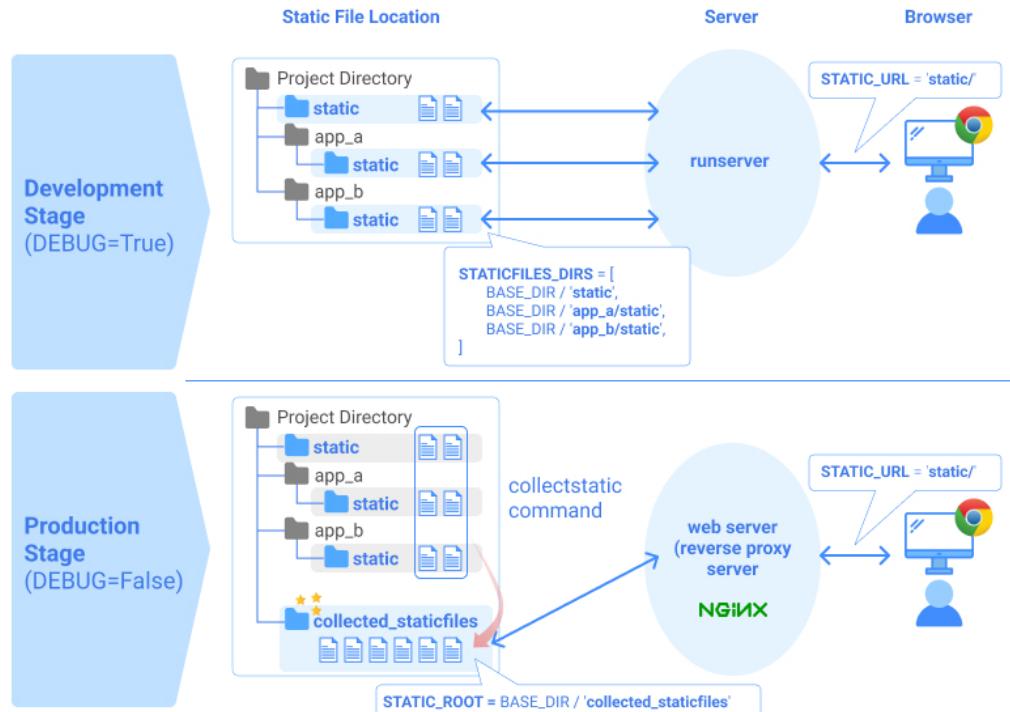
```
SECRET_KEY=django-insecure-a6e%75-^!.....
ALLOWED_HOSTS=localhost,xx.xx.xx.xx
```

```
DB_ENGINE=django.db.backends.postgresql
DB_NAME=project_d
DB_USER=project_d_user
DB_PASSWORD=project_d_pass
DB_HOST=localhost
DB_PORT=5432
```

There are some points you need to be careful about.

- **no " (quotation)** for all settings
- For **ALLOWED_HOSTS**, you need to put the static IP address that is attached to the Ubuntu instance. Keep **localhost** for testing purposes. When you write more than two hosts, make sure that there are **no [] (brackets)** and **no space**
- For database settings, make sure that the values are the same as the ones you set when you created the database.

Static File Settings



Setting `STATIC_ROOT` is one of the configurations that can often create an error as the mechanism of statics file handling is often incorrectly understood.

Static file locations

Django is designed to let a **web server** or a **reverse proxy server** (e.g., [Nginx](#)) handle static files directly in the production environment. By this configuration, the Django app can focus more on the dynamic part.

Development Stage

When `DEBUG` is `True`, Django considers that the app is running still under the development environment, and the development server (**runserver**) gets the static files from the original locations. The locations are specified through the `STATICFILES_DIRS` settings. You

can list multiple locations. Usually, there are two types of locations. One is directly under the project file. The other is under each app directory.

Assuming `STATIC_URL = 'static/'`, when an HTTP request with `/static/` comes, the development server handles the static files located in the static directories specified by `STATICFILES_DIRS`.

Production Stage

When `DEBUG` is switched to `False`, Django considers that the app is running under the production environment and lets the **web server** (or reverse proxy server) handle the static files stored in the specified location set by `STATIC_ROOT`.

Assuming `STATIC_URL = 'static/'` is written in the `settings.py` file, when an HTTP request with `/static/` comes, the web server handles the static files located in the `static` directory specified by `STATIC_ROOT`.

collectstatic

The `collectstatic` command collects all static files located in several places and puts them in the directory specified by `STATIC_ROOT`.

If you don't do this, your web application UI will be broken.

This is an example of the Django admin page when you forget to run the `collectstatic` command. We'll demonstrate the command in the next section.

Django administration

Welcome, bloovee. [View site](#) / [Change password](#) /
[Log out](#)

Site administration

[Accounts](#)

[Email addresses](#) [Add](#) [Change](#)

[Authentication and](#)
[Authorization](#)

[Groups](#) [Add](#) [Change](#)

[Users](#) [Add](#) [Change](#)

[Employee Learning](#)

[Divisions](#) [Add](#) [Change](#)

[Employees](#) [Add](#) [Change](#)

[Learning courses](#) [Add](#) [Change](#)

[Personal infos](#) [Add](#) [Change](#)

Tips: How to write STATIC_ROOT

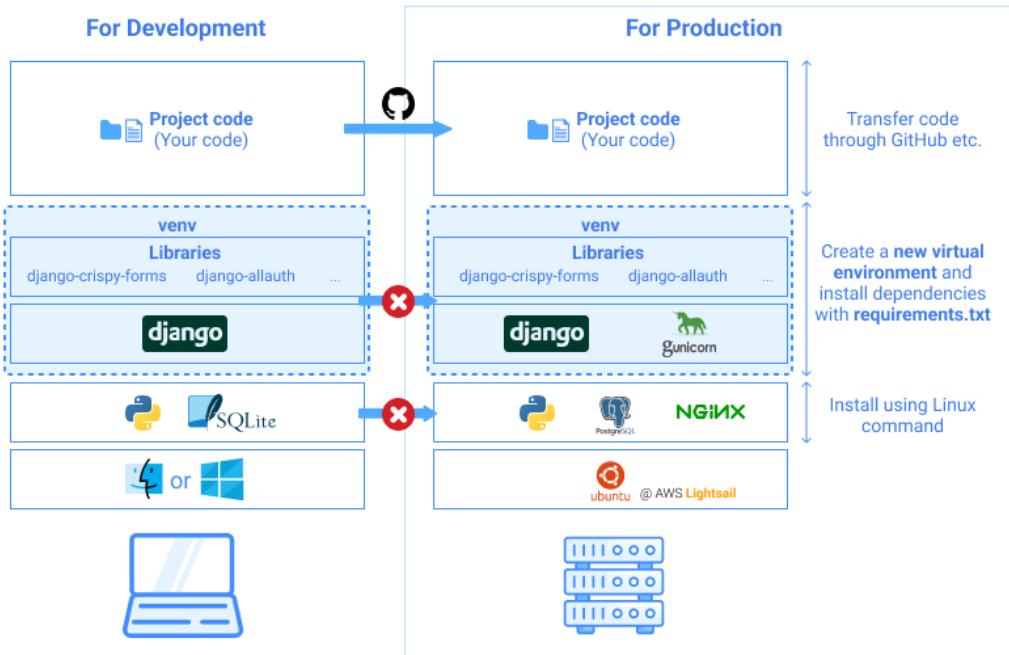
For a beginner, writing `STATIC_ROOT` can be confusing. As you can set multiple locations for `STATICFILES_DIRS`, you should use a list format for `it`. However, as `STATIC_ROOT` should have only one directory path, you should not use the list format.

```
STATIC_URL = 'static/'  
  
STATICFILES_DIRS = [ BASE_DIR / 'static' ] ← should be list  
  
✓ STATIC_ROOT = BASE_DIR / 'collectstatic_path' ← should Not be list
```

```
STATIC_URL = 'static/'  
  
STATICFILES_DIRS = [ BASE_DIR / 'static' ] ← should be list  
  
✗ STATIC_ROOT = [ BASE_DIR / 'collectstaticf_path' ] ← should Not be list
```

Also, the `STATIC_ROOT` path should not be the same as the one in `STATICFILES_DIRS`, as the static files under `STATICFILES_DIRS` will be copied to `STATIC_ROOT` by the `collectstatic` command. We'll explain how to set `STATIC_ROOT` in our app on the next page along with the `collectstatic` command.

Django and Dependency Installation on Production Server



If you are a beginner, you may not know how to build the production environment and which code should be transferred from the local computer to the production server. There are four groups of code layers in the Django app.

1. **OS**: On the local computer, you may be using Mac OS or Windows. On the server, LinuxOS is the most popular choice. We have already prepared *Ubuntu OS* in *AWS Lightsail* at the beginning of this chapter.
2. **Programs directly installed onto the production server**: Python3, database software (e.g., PostgreSQL), and web server software (e.g., Nginx) should be directly installed onto the production server. For Ubuntu OS, use `apt` or `apt-get` command for installation.

3. **Programs installed in the virtual environment:** Django itself, gunicorn and other Django libraries should be installed in the virtual environment to make sure the consistency of dependencies. Use the `venv` command to create another virtual environment on the production server first, and run the `pip` command with `requirement.txt`.
4. **Project code (your code):** This is the code you developed on the local computer. We have already explained how to transfer the code to the production server using *GitHub* at the beginning of this chapter.

This lesson will cover the third step with a quick check of the Python version.

Confirm Python3 is installed on the instance

To check if Python 3 is already installed in your instance, run the `python3 --version` command.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | python3 --version
```

Command Line - RESPONSE

```
Python 3.8.10
```

If it's not installed, you can install the latest available version on Ubuntu by running the command below.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo apt update  
ubuntu@ip-xx:~$ | sudo apt install python3
```

Install and create a virtual environment

To replicate the same environment as the one on your local computer, create a virtual environment first.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo apt install -y python3-venv
```

Go to the project directory and run the `venv` command. We use `d_env` as a virtual environment name.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | cd ~/project_d  
ubuntu@ip-xx:~$ | python3 -m venv d_env
```

Activate the virtual environment using the `source` command.

Command Line - INPUT

```
ubuntu@ip-xx:~/project_d$ | source d_env/bin/activate
```

Prepare a requirements file for production

As additional packages need to be installed for production, we must add them to the `requirements.txt` file. There is a technique to structure the requirement files for development and production nicely. We will explain it later in this chapter.

On top of the current listed dependencies, you need to add:

- **gunicorn** (application server)
- **psycopg2-binary** (adapter of PosgreSQL)

requirements.txt

```
Django==4.1.7  
django-crispy-forms  
crispy-bootstrap5  
django-allauth  
django-environ  
gunicorn  
psycopg2-binary
```

Install Django and dependencies

Finally, you can install Django and dependencies by running the `pip` command using `requirements.txt`.

Command Line - INPUT

```
ubuntu@ip-xx:~/project_d$ | pip install -r requirements.txt
```

If you successfully installed them, you'll see a message like the one below.

Command Line - RESPONSE

```
:  
Successfully installed Django-4.1.7 asgiref-3.6.0  
backports.zoneinfo-0.2.1 certifi-2022.12.7 cffi-1.15.1 charset-  
normalizer-3.1.0 crispy-bootstrap5-0.7 cryptography-40.0.2  
defusedxml-0.7.1 django-allauth-0.54.0 django-crispy-forms-2.0  
django-environ-0.10.0 gunicorn-20.1.0 idna-3.4 oauthlib-3.2.2  
psycopg2-binary-2.9.6 pycparser-2.21 pyjwt-2.6.0 python3-openid-  
3.2.0 requests-2.28.2 requests-oauthlib-1.3.1 sqlparse-0.4.4  
urllib3-1.26.15
```

Run the Django setup commands

No database is migrated at this stage, and no superuser is created.
Run the as-usual commands on the server.

Command Line - INPUT

```
ubuntu@ip-xx:~/project_d$ | python manage.py makemigrations  
ubuntu@ip-xx:~/project_d$ | python manage.py migrate  
ubuntu@ip-xx:~/project_d$ | python manage.py createsuperuser
```

Note: Database Owner

When you run the migrate command, you may encounter an error like the one shown below.

Command Line - RESPONSE

```
...django.db.migrations.exceptions.MigrationSchemaMissing: Unable  
to create the django_migrations table (permission denied for schema  
public  
LINE 1: CREATE TABLE "django_migrations" ("id" bigint NOT NULL  
PRIMA...
```

This may be because of the database owner setting for PostgreSQL. This setting was not required in the older version of PostgreSQL, but it is required in the recent versions. To set it up, run the command below after entering the PostgreSQL prompt.

Command Line - INPUT

```
[user@localhost] | ALTER DATABASE project_d OWNER TO  
project_d_user;
```

When it is successful, the command line returns the message below.

Command Line - RESPONSE

ALTER DATABASE

Set STATIC_ROOT and run the collectstatic command

For the production environment, you need to set the static file location (`STATIC_ROOT`) and collect static files by running the `collectstatic` command.

In our case example, set the directory in `BASE_DIR / 'collectstatic_path'` for now. Add the setting in the settings file. This setting will be adjusted when we establish the *Nginx* server.

config/settings/production.py

```
:  
STATIC_ROOT = BASE_DIR / 'collectstatic_path'
```

Then, run the `collectstatic` command.

Command Line - INPUT

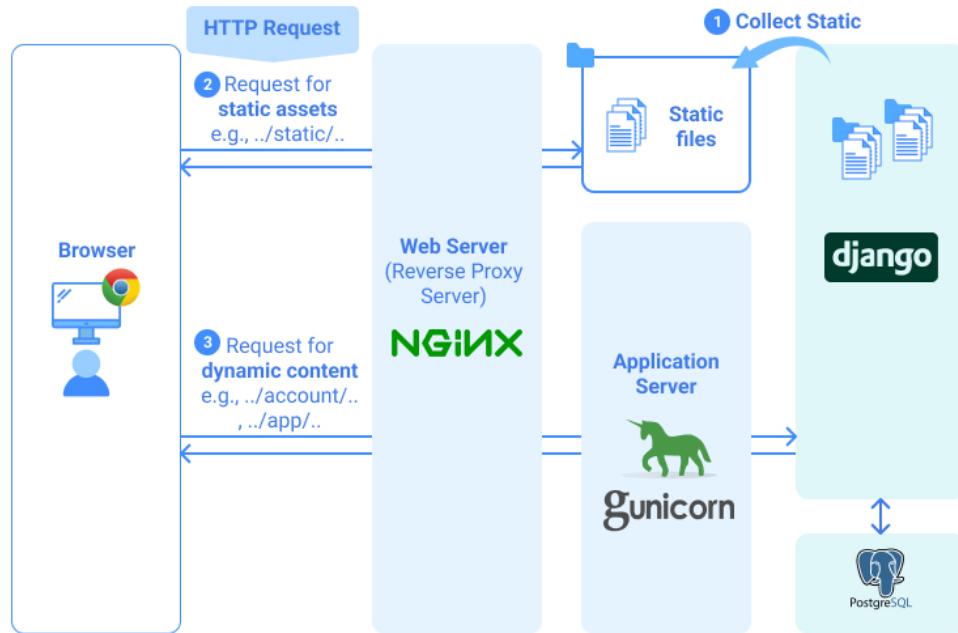
```
(d_env) ubuntu@ip-xx:~/project_d$ | python manage.py collectstatic
```

If there is no error, you'll see the message like the one below.

Command Line - RESPONSE

```
138 static files copied to  
'/home/ubuntu/project_d/collectstatic_path'.
```

Web Server and Application Server in Django



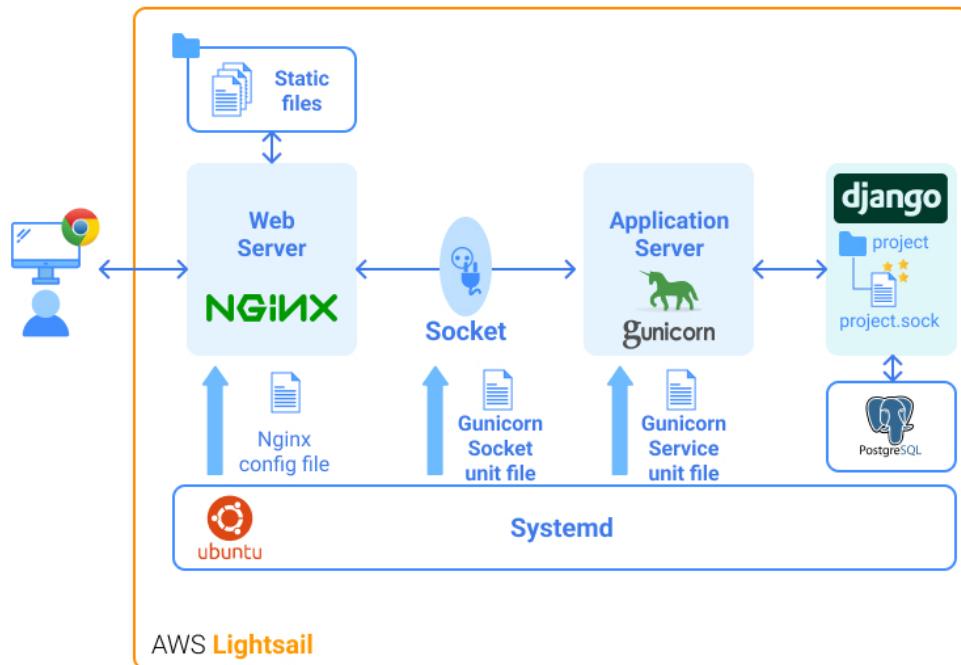
As explained in the beginning of this chapter, two types of server functionalities are required for dynamic web applications – a **web server** and **application server**. The key difference between those two servers is that a web server manages static content while an application handles dynamic content.

The main figure illustrates a use case in Django applications using **Nginx** as a web server and **Gunicorn** as an application server. As Nginx handles all requests from clients first, it is also called a **reverse proxy server**.

The mechanism is as follows:

1. Using the `collectstatic` command, all static files are stored in a particular location that Nginx can also recognize. (This is important for Nginx configurations. We'll explain this later.)
2. When an HTTP request for static assets comes, Nginx handles the request directly. The static URL is defined in the `settings.py` (`STATIC_URL`).
3. When an HTTP request is not for static assets (request for dynamic content), the request goes through Gunicorn, and the Django app handles the request using the **URL dispatcher**, **Views** and **Models** etc.

Application Server Setup – Gunicorn



The **Gunicorn** (Green Unicorn) is a Python Web Server Gateway Interface (**WSGI**) HTTP server. As we have already installed Gunicorn, we'll explain how to set it up in this section. To manage the Gunicorn application server on a Linux server, we can use **systemd** to run the server.

As web services should always be available on the Internet, the process must run when the server runs. **systemd** is a daemon (or a service) that manages other **daemons** for Linux OS. When Linux OS starts or reboots, systemd also starts and continues running to centrally control other daemons until the OS shuts down.

If you want to learn more about systemd, please check our '**Linux OS Introduction**' course.

[Linux OS Introduction 'Systemd'](#)

There are three steps for Gunicorn setup.

1. To run *Gunicorn* using *systemd*, you need to create two **unit files**. Unit files are used to set rules to manage units (processes) by *systemd*.
2. Enable the **unit files** and start the units
3. Check if the **units** are running

Create unit files

To run Gunicorn, you need to create a **service unit file** and a **socket unit file**.

Create a service unit file

A service unit file is used to create a service to run the application itself. The file should be stored under */etc/systemd/system*. You can set any name but usually use a project name like *project_d.service*. To create and edit the file, use the *vim* command.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo vim /etc/systemd/system/project_d.service
```

Edit the file like in the example below.

```
/etc/systemd/system/project_d.service
[Unit]
Description=gunicorn daemon
Requires=project_d.socket
After=network.target

[Service]
User=ubuntu
Group=www-data
WorkingDirectory=/home/ubuntu/project_d
ExecStart=/home/ubuntu/project_d/d_env/bin/gunicorn \
--access-logfile - \
--workers 3 \
--bind unix:/home/ubuntu/project_d/project_d.sock \
config.wsgi:application

[Install]
WantedBy=multi-user.target
```

You need to carefully type file paths, as the file paths are also used in other files. For example, *home/ubuntu/project_d/project_d.sock* is the actual file that will be created under the project directory when the socket is enabled. The same path should be written in the **socket**

unit file and the **Nginx configuration file**, which will be explained in the next section.

Create a socket unit file

A socket unit file is used to establish socket communication between Gunicorn and Nginx. To create and edit the file, use the `vim` command.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo vim /etc/systemd/system/project_d.socket
```

Edit the file like in the example below.

```
/etc/systemd/system/project_d.socket
```

```
[Unit]
Description=gunicorn socket

[Socket]
ListenStream=/home/ubuntu/project_d/project_d.sock

[Install]
WantedBy=sockets.target
```

Enable the unit files and start the units

You can enable and start the service and socket by running the command below. The `systemctl enable` command is used to enable units. With the `--now` option, you can start the unit at the same time. You can omit `.service` when you run the command for a service unit.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo systemctl enable --now project_d
ubuntu@ip-xx:~$ | sudo systemctl enable --now project_d.socket
```

Check if the units are running

To check the socket unit status, run the `systemctl status` command.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | systemctl status project_d.socket
```

If the socket is properly running, you can see the message like the one below.

Command Line - RESPONSE

```
● project_d.socket - gunicorn socket
Loaded: loaded (/etc/systemd/system/project_d.socket; enabled;
vendor preset: enabled)
Active: active (running) since Fri 2023-04-21 11:55:33 UTC; 22h ago
Triggers: ● project_d.service
Listen: /home/ubuntu/project_d/project_d.sock (Stream)
Tasks: 0 (limit: 2372)
Memory: 0B
CGroup: /system.slice/project_d.socket
```

To check the service unit status, run the command below.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | systemctl status project_d
```

If the service is running properly, you can see the message like the one below.

Command Line - RESPONSE

```
● project_d.service - gunicorn daemon
Loaded: loaded (/etc/systemd/system/project_d.service; disabled;
vendor preset: enabled)
Active: active (running) since Fri 2023-04-21 16:05:41 UTC; 18h ago
TriggeredBy: ● project_d.socket
Main PID: 123874 (gunicorn)
Tasks: 4 (limit: 2372)
Memory: 119.5M
CGroup: /system.slice/project_d.service
```

To learn about the unit files for Gunicorn, you can check the Gunicorn official documentation.

[Gunicorn documentation reference: Deploying Gunicorn systemd](#)

Note: Check Gunicorn process

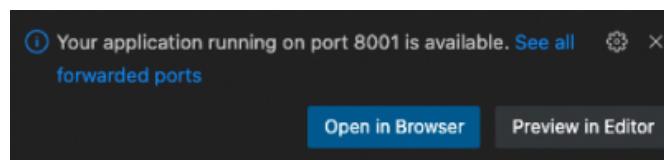
We checked if the socket and service units were running, but we didn't check if the application was running using Gunicorn. Here is a way to check if Gunicorn is working.

Run the command below after activating the virtual environment. We use port `8001` in case port `8000` is used for the local app.

Command Line - INPUT

```
(d_env) ubuntu@ip-xx:~/projects_d$ | gunicorn --bind=0.0.0.0:8001 config.wsgi:application
```

If Gunicorn starts appropriately, you'll see a pop-up in the VS Code window like the one below. Click the **Open in Browser** button to see the app.

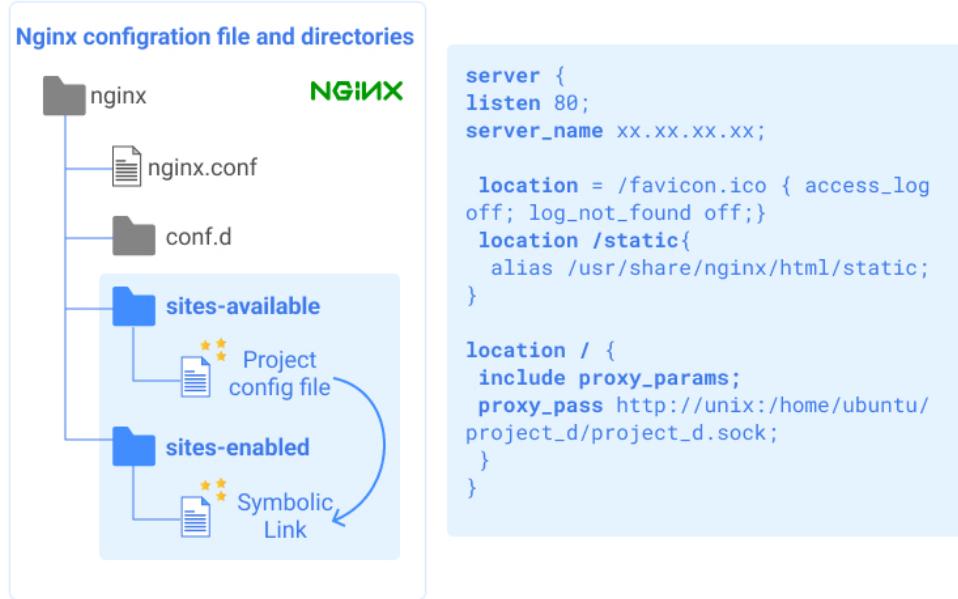


Go to the admin page. You'll see that the app is running, but CSS has not been properly applied yet.

Django administration

Username:
Password:

Web Server Setup – Nginx



Nginx is one of the most popular web servers. With a focus on static content handling, Nginx can be more scalable and faster than other web servers. This lesson will explain how to set up Nginx for Django app deployment.

There are five steps to set up Nginx:

- 1. Install Nginx**
- 2. Add an Nginx configuration file to sites-available**
- 3. Add a symbolic link of the configuration file to sites-enabled**
- 4. Update static file location**
- 5. Adjust firewall settings**
- 6. Check if the Django app is running**

Note: For the Nginx setting, the virtual environment is not needed as it's directly installed on the OS.

Install Nginx

Install Nginx using the `apt-get` command.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo apt-get update  
ubuntu@ip-xx:~$ | sudo apt-get install -y nginx
```

As Nginx runs when it is installed, check if it is running by the `systemctl status` command.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | systemctl status nginx
```

Unless you encounter an issue, you'll see the message below.

Command Line - RESPONSE

```
● nginx.service - A high performance web server and a reverse proxy server  
  Loaded: loaded (/lib/systemd/system/nginx.service; enabled;  
           vendor preset: enabled)  
  Active: active (running) since Fri 2023-04-21 16:05:47 UTC; 21h ago  
    Docs: man:nginx(8)  
   Process: 123945 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on;  
             master_process on; (cod>  
             :  
             :
```

Add an Nginx configuration file

Nginx has several configuration setting points.

1. **nginx.conf file:** the original settings are written here. Usually, we don't touch this file.
2. **conf.d directory:** you can save your custom configuration file in this directory. The `nginx.conf` file will include your custom configuration. As the configuration files saved in this directory are

always read by Nginx, there is limited flexibility to switch configurations.

3. **sites-available directory:** this is another directory where you can save your custom configuration files; however, Nginx does not read the files directly. To enable the configuration, you need to create a symbolic link of the file in the *sites-enabled* directory. By following this rule, you can easily switch configuration files by controlling symbolic links.
4. **sites-enabled directory:** the location where you can put the symbolic link of the configuration file that was created in the *sites-available* directory.

To create a new configuration file, run the command below.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo vim /etc/nginx/sites-available/project_d
```

Edit the file like in the example below.

```
/etc/nginx/sites-available/project_d
```

```
server {  
    listen 80;  
    server_name xx.xx.xx.xx;  
  
    location = /favicon.ico { access_log off; log_not_found off; }  
    location /static{  
        alias /usr/share/nginx/html/static;  
    }  
  
    location / {  
        include proxy_params;  
        proxy_pass http://unix:/home/ubuntu/project_d/project_d.sock;  
    }  
}
```

There are three key parts that you need to type carefully.

1. **server name:** the **static IP address** attached to the Ubuntu instance
2. **location /static:** the address should be aligned with **STATIC_ROOT**
3. **proxy_pass:** the path of the *sock* file defined in the Gunicorn service and socket unit files in the previous section

Add a symbolic link and verify the syntax

To enable the new settings, you need to create a symbolic link of the file in the sites-enabled directory. Run the command below to create the symbolic link.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo ln -s /etc/nginx/sites-available/project_d  
/etc/nginx/sites-enabled
```

Use the Nginx command to check if the configuration file is aligned with the Nginx syntax.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo nginx -t
```

If the configuration file is properly written, you'll see the message below.

Command Line - RESPONSE

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok  
nginx: configuration file /etc/nginx/nginx.conf test is  
successful
```

Update static file location

Update the static file location to the directory that Nginx usually uses. To update the location, edit STATIC_ROOT in the settings file and run the `collectstatic` command.

Update `STATIC_ROOT` to the directory path we used in the Nginx configuration file.

sconfig/settings/production.py

```
:  
STATIC_ROOT = '/usr/share/nginx/html/static'
```

Create the `static` directory and change the owner to `ubuntu` (the user of the instance). Then, run the `collectstatic` command under the virtual environment.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo mkdir /usr/share/nginx/html/static  
ubuntu@ip-xx:~$ | sudo chown -R ubuntu  
/usr/share/nginx/html/static  
ubuntu@ip-xx:~$ | cd ~/project_d  
ubuntu@ip-xx:~$ | source d_env/bin/activate
```

Command Line - INPUT

```
(d_env) ubuntu@ip-xx:~/project_d$ | python manage.py  
collectstatic
```

You'll see that the static files are successfully copied to the new directory.

Command Line - RESPONSE

```
138 static files copied to '/usr/share/nginx/html/static'
```

As the old static file directory is not needed anymore, delete it.

Command Line - INPUT

```
(d_env) ubuntu@ip-xx:~/project_d$ | p rm -r collectstatic_path
```

Firewall settings

This process is optional as AWS Lightsail has its firewall. On Ubuntu OS, you can use UFW (Uncomplicated Firewall) to manage *iptables*.

To enable UFW, run the command below.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo ufw enable
```

You'll see the message below. This explains the importance of opening an SSH port to keep the SSH connection.

Command Line - INTERACTIVE

```
Command may disrupt existing ssh connections. Proceed with  
operation (y|n)? y  
Firewall is active and enabled on system startup
```

Open the SSH port and ports for Nginx by running the following commands.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo ufw allow ssh  
ubuntu@ip-xx:~$ | sudo ufw allow 'Nginx Full'
```

Run the status command to check the status.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo ufw status
```

You can see that the UFW is active now with two allowed ports settings.

Command Line - RESPONSE

```
Status: active  
To           Action      From  
--           -----      ---  
22           ALLOW       Anywhere  
Nginx Full   ALLOW       Anywhere
```

For more details about UFW, please check our '[Linux OS Introduction](#)' course.

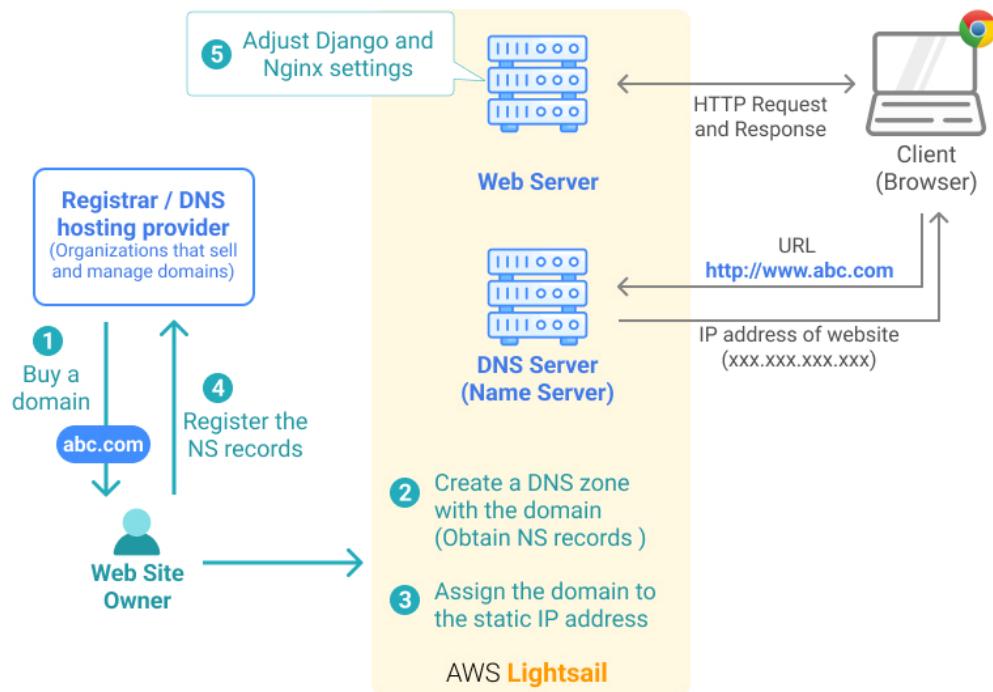
[Linux OS Introduction 'UFW \(Uncomplicated Firewall\)'](#)

Check if the app is running

If all settings are properly done, you'll be able to see your app through the static IP address.

The screenshot shows a web application titled "Digital Training Registration". At the top, there is a navigation bar with links for "Training Registration", "List", "Create", "Sign in", and "Sign Up". Below the navigation, the title "Digital Training Registration" is centered. A message "As of April 24, 2023" is displayed. Two blue buttons at the bottom are labeled "View Registered Programs" and "Create a New Program". At the very bottom of the page, a small footer note reads "2023 Employee Learning".

Domain Setup



Accessing your web app using only an IP address is not user-friendly. To make your web app accessible through your domain, register your domain and attach the domain to the IP address.

There are five steps to complete this process to set up a domain for your app on the Lightsail platform.

- 1. Buy and register your own domain**
- 2. Create a DNS zone in the Lightsail console (Obtain NS records)**
- 3. Assign the domain to the static IP address in the Lightsail console (Adding an A record to the DNS zone)**
- 4. Register the NS records at the DNS hosting provider**
- 5. Adjust Django and Nginx settings**

Buy and register your own domain

If you don't have your domain, you must buy it from a **registrar** who manages and sells domains. As registrars often manage and host domains on their **DNS servers**, they are also called **DNS hosting providers**.

There are many providers. You can choose any of them. Amazon also provides a domain service called **Route 53**.

Usually, domains with '.com' are less available and more expensive. Cheap domains are available for less than \$10 per year. For practice purposes, a cheap one is enough, but you need to make sure it won't be auto-renewed, or else you may get unexpected charges in the future.

Create a DNS zone in the Lightsail console (Obtain NS records)

Once you get your own domain, you can register it and create a **DNS zone** in the Lightsail console. In the following steps, we assume that you bought a domain other than Amazon Route 53.

From the home page (not from the instance page), select the **Domain & DNS** tab. Click on the **Create DNS zone** button.

Good morning!

Instances Containers Databases Networking Storage Domains & DNS Snapshots

Create a DNS Zone

A DNS zone contains the domain name system (DNS) records for your domain. By creating a DNS zone, you can easily map your registered domain and its subdomains to your Lightsail resources. If you have a domain registered elsewhere, you can add it to Lightsail by creating a DNS zone.

[Learn more about DNS zones](#)

[Create DNS zone](#)

Register a domain

Use Lightsail to register a new domain with Amazon Route 53. After you register your domain name, Lightsail automatically creates a DNS zone that has the same name as the domain.

[Learn more about domain registration in Lightsail](#)

[Register domain](#)

Add your own domain.

 Create a DNS Zone

A DNS zone contains the domain name system (DNS) records for your domain. Create a DNS zone for your registered domain to map it and its subdomains to your Lightsail resources, such as an instance or load balancer.

[Learn more about DNS in Amazon Lightsail](#)

 DNS zones are **Global** resources.
They can reference any instance or load balancer in any Region.

Domain configuration

You must register your domain before creating a DNS zone.

[Learn more about domain registration in Lightsail](#)

Domain source

Use a domain that is registered with Amazon Route 53
 Use a domain from another registrar

Domain name

Specify your registered domain name.

example.com

 Enter the first part of the name and the extension (such as example.com), without www.

Confirm that a DNS zone is created and check the **NS records**. You'll need to add these records to the DNS hosting provider's website.

The screenshot shows the AWS Lightsail DNS zone management interface for the domain 'example.com'. The top navigation bar includes tabs for 'Domains' (which is selected), 'Assignments', and 'DNS records'. Below the navigation is a section titled 'Domain' with the sub-instruction 'Use this DNS zone to manage your domain.' A callout box contains the note: 'Update the name servers of your domain to match the name servers of this DNS zone.' In the 'Name servers' section, it states that Lightsail assigns name servers when creating a DNS zone. It also provides instructions for configuration via Lightsail's DNS service or domain registration. A link 'Learn more about name server records' is provided. A list of four name servers is shown, each preceded by 'ns-' and followed by a placeholder IP address and a suffix: '.awsdns-52.org', '.awsdns-03.net', '.awsdns-00.co.uk', and '.awsdns-14.com'. The entire list is highlighted with an orange rectangular box.

Assign the domain to the static IP address in the Lightsail console

To match your domain name and static IP address, assign the domain to the static IP address. This action is the same as adding an **A record** to the DNS.

For the domain, you can also use a **subdomain**.

The screenshot shows the 'Assignments' tab of the Lightsail domain configuration interface. At the top, it displays 'example.com' with a network icon, 'DNS zone', and 'Global, all zones'. Below this, there's a section titled 'Domain assignments' with a brief description: 'Use assignments to point a domain to your Lightsail resources, such as load balancers and instances.' A link 'Learn more about domain assignments' is provided. A sub-section titled 'Assign this domain to a resource.' includes a '+ Add assignment' button. The main form fields are: 'Select a domain name' (radio button selected for 'example.com'), 'Select a resource' (dropdown menu showing 'Staticip-1'), and 'Select the address' (radio button selected for 'Static IP address'). A note below states: 'employee-learning.site will resolve to Staticip-1, which is attached to this instance.' At the bottom right are 'Cancel' and 'Assign' buttons.

After assigning it, you can check the **DNS record** tab. You can see that the A record has been added.

The screenshot shows the 'DNS records' tab of the Lightsail domain configuration interface. At the top, it displays 'example.com' with a network icon, 'DNS zone', and 'Global, all zones'. Below this, there's a section titled 'DNS records' with a brief description: 'Each record in a DNS zone defines how you want to route internet traffic for your domain. For example, you can add DNS records that route traffic to your Lightsail resources, another domain, or a mail server.' A link 'Learn more about editing DNS records' is provided. A sub-section titled 'A RECORDS' includes a '+ Add record' button. The main table lists one record: 'Record name' is 'example.com', 'Route traffic to' is 'XX.XX.XX.XX', and there are edit and delete icons to the right.

Register the NS records at the DNS hosting provider

Next, you need to go back to the DNS hosting provider's web site to add the NS records. Usually, you can register NS records on the account pages of hosting providers. Login with your account and find a page where you can add your NS records.

User interfaces differ by provider but you need to make sure that all NS records are registered like shown in the image below.

The screenshot shows a 'DNS & NAMESERVERS' page for a domain named 'EMPLOYEE-LEARNING.SITE'. On the left, there's a sidebar with sections for Summary, Builders (WebsiteBuilder, WordPress), Email (Google Workspace), Security (SSL Certificate, SiteLock), Advanced (Pointers & Subdomains), DNS & Nameservers (selected), Transfers, and Contact Information. The main content area has tabs for NAMESERVERS, PRIVATE NAMESERVERS, DNS RECORDS, and CONNECTED APPS & WEBSITES. Under NAMESERVERS, it says 'Every domain requires a set of nameservers to allow visitors on the Internet to reach it. Use caution when editing your Nameservers.' Below this, it says 'Changes may take 48 hours to propagate.' There's a blue button labeled '+ Add Nameserver' and a 'Restore Default' button. A list of nameservers is shown: ns1.domain.com, ns2.domain.com, ns-1446.awsdns-52.org, ns-539.awsdns-03.net, ns-1542.awsdns-00.co.uk, and ns-112.awsdns-14.com, each with a three-dot menu icon to its right.

It may take time to update the name server information.

Adjust Django and Nginx settings and restart

To make your web app accessible by the domain, you need to add the domain information in Django settings and Nginx configurations.

Add the domain to Django settings

You need to add the domain to `ALLOWED_HOSTS`. Open the .env file to edit it.

.env

```
ALLOWED_HOSTS=localhost,xx.xx.xx.xx,example.com
```

Add the domain to Nginx configurations

You need to add the domain to `server_name`. Open `/etc/nginx/sites-available/project_d` to edit it. To add a domain, you don't need to use ,(comma). Instead, use a space after the static IP address.

`/etc/nginx/sites-available/project_d`

```
server {  
    listen 80;  
    server_name xx.xx.xx.xx example.com;  
  
    location = /favicon.ico { access_log off; log_not_found off; }  
    location /static{  
        alias /usr/share/nginx/html/static;  
    }  
  
    location / {  
        include proxy_params;  
        proxy_pass http://unix:/home/ubuntu/project_d/project_d.sock;  
    }  
}
```

Restart Gunicorn and Nginx

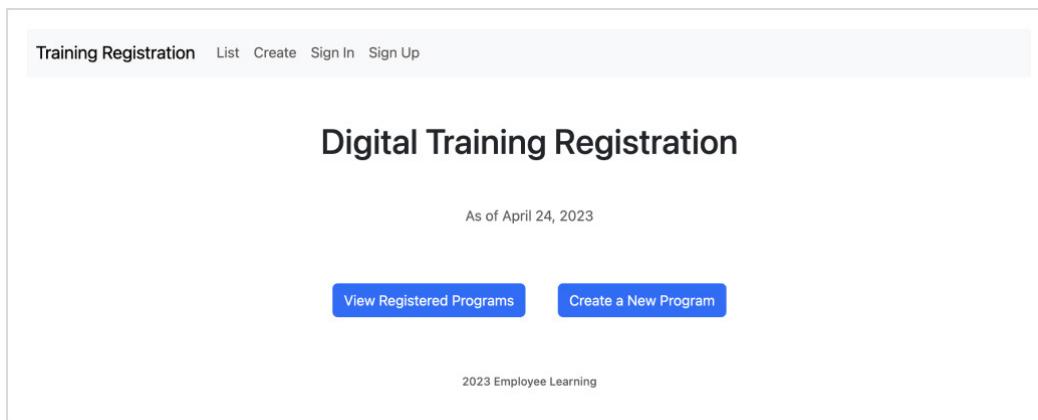
To reflect the settings, restart Nginx.

Command Line - INPUT

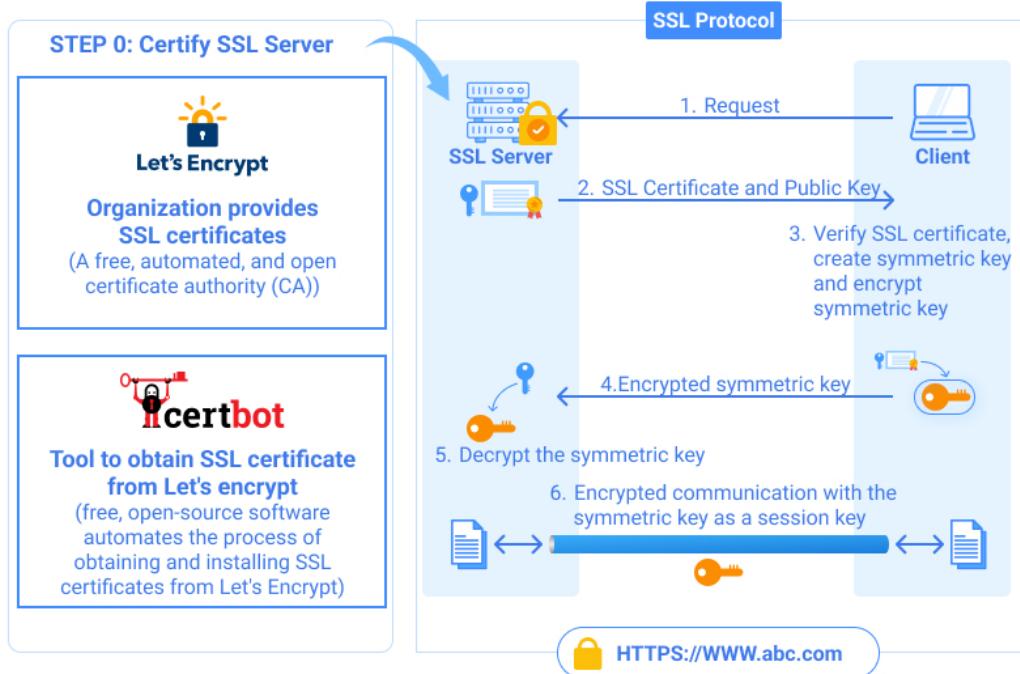
```
ubuntu@ip-xx:~$ | sudo systemctl restart project_d  
ubuntu@ip-xx:~$ | sudo systemctl restart nginx
```

Check if the app is accessible through the domain

If all settings are properly done, you'll be able to see your app through the domain now.



SSL Setup – Certbot



SSL (Secure Sockets Layer) is a protocol used to establish secure and encrypted communication over the Internet. Most websites use the **HTTPS** (HyperText Transfer Protocol Secure) protocol to increase their security level. SSL is necessary for the HTTPS protocol, which uses a combination of the standard HTTP protocol and the SSL protocol.

To establish SSL communication, the web server has to be certified by a certificate authority (CA). **Let's encrypt** is one of the CAs and provides SSL certificates for free. To obtain a Let's encrypt SSL certificate, **Certbot** is often used. It is free, open-source software that automates the process of obtaining and installing SSL certificates from Let's Encrypt.

SSL authentication mechanism

As explained, a prerequisite for establishing SSL communication is having a web server with an SSL certificate (SSL server). Here are the key processes of the SSL authentication mechanism:

1. A client sends a request to the SSL server
2. When the SSL server receives the request, the SSL server sends its SSL certificate with its *public key* to the client
3. Once the client receives the SSL certificate and verifies it, the client creates a symmetric key and encrypts the *symmetric key* using the *public key* from the server
4. Then, the client sends the encrypted *symmetric key* to the SSL server
5. When the SSL server receives the encrypted *symmetric key*, the SSL server decrypts it with its own *private key*. By now, the client and the SSL server have the same *symmetric key* sent through a secured channel.
6. Both use the *symmetric key* as a session key to create a secured communication channel

As you can see in the processes above, SSL uses both **Asymmetric Encryption** and **Symmetric Encryption**. SSL uses Asymmetric Encryption for the first authentication, which is more secure. For a session key, SSL uses Symmetric Encryption, which is faster.

How to set up SSL using Certbot

Certbot allows you to get an SSL certificate on your web server easily and quickly.

Here are the key steps:

-
1. Install Certbot for Nginx on Ubuntu OS
 2. Open an SSL port
 3. Obtain an SSL certificate by running the certbot command

4. Check the result and test the configuration

Install Certbot for Nginx on Ubuntu OS

Run the command below if you are using Nginx.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo apt-get install certbot python3-certbot-nginx
```

If you are using a different server, check the Certbot instruction page.

Certbot Instructions

Open SSL port

By default, the SSL port (HTTPS port) is not open in AWS Lightsail. Go to the Lightsail console and check the **Networking** tab under the Ubuntu Instance.

The screenshot shows the AWS Lightsail instance details for an Ubuntu-1 instance. The Networking tab is active. In the IPv4 networking section, it shows a static public IP (XX.XX.XX.XX) and a private IP (172.26.8.65). The IPv4 Firewall section contains two entries:

Application	Protocol	Port or range / Code	Restricted to
SSH	TCP	22	Any IPv4 address Lightsail browser SSH/RDP
HTTP	TCP	80	Any IPv4 address

Select **HTTPS** and click on the **Create** button.

The screenshot shows the 'IPv4 Firewall' configuration page. A new rule is being added for port 443. The 'Protocol' dropdown is set to 'TCP'. The 'Port or range' field contains '443'. The 'Restricted to' section shows 'Any IPv4 address'. The 'Create' button is highlighted in green with a checkmark, indicating the rule has been successfully created. Other ports listed are 22 (SSH) and 80 (HTTP).

Application	Protocol	Port or range	Restricted to
Custom	TCP	443	Any IPv4 address
All TCP			
All UDP			
All protocols			
Instance access			
SSH		22	Any IPv4 address Lightsail browser SSH/RDP
RDP		80	Any IPv4 address
Web server		587	Any IPv4 address
HTTP			
HTTPS			

Now, the **443** port (for HTTPS) becomes open.

The screenshot shows the 'IPv4 Firewall' configuration page with three rules listed: SSH (port 22), HTTP (port 80), and HTTPS (port 443). All three rules have 'Any IPv4 address' as the restricted IP and include the 'Lightsail browser SSH/RDP' note.

Application	Protocol	Port or range / Code	Restricted to
SSH	TCP	22	Any IPv4 address Lightsail browser SSH/RDP
HTTP	TCP	80	Any IPv4 address
HTTPS	TCP	443	Any IPv4 address

If you are also using **UFW**, check the status by running the `ufw status` command. '**Nginx Full**' (or port 443) should be allowed for SSL communication.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo ufw status
```

Command Line - RESPONSE

```
Status: active
To          Action        From
--          -----       ----
```

22/tcp	ALLOW	Anywhere
NginxFull	ALLOW	Anywhere

Obtain an SSL certificate by running the certbot command

Run the command below to obtain an SSL.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo certbot --nginx -d your_domain
```

When you run the command, several questions are asked. Here are the questions and answers for our case.

Your email

Command Line - INTERACTIVE

```
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Plugins selected: Authenticator nginx, Installer nginx
Enter email address (used for urgent renewal and security notices) (Enter 'c' to cancel): YOUR EMAIL
```

Terms of Services

Command Line - INTERACTIVE

```
-----  
Please read the Terms of Service at  
https://letsencrypt.org/documents/LE-SA-v1.3-September-21-2022.pdf. You must  
agree in order to register with the ACME server at  
https://acme-v02.api.letsencrypt.org/directory  
-----  
(A)gree/(C)ancel: A
```

You information disclosure

Command Line - INTERACTIVE

```
-----  
Would you be willing to share your email address with the  
Electronic Frontier  
Foundation, a founding partner of the Let's Encrypt project and  
the non-profit  
organization that develops Certbot? We'd like to send you email  
about our work  
encrypting the web, EFF news, campaigns, and ways to support  
digital freedom.
```

```
- - - - -  
(Y)es/(N)o: N
```

Redirection of HTTP request

Command Line - INTERACTIVE

```
Please choose whether or not to redirect HTTP traffic to HTTPS,  
removing HTTP access.  
- - - - -  
1: No redirect - Make no further changes to the webserver  
configuration.  
2: Redirect - Make all requests redirect to secure HTTPS access.  
Choose this for  
new sites, or if you're confident your site works on HTTPS. You  
can undo this  
change by editing your web server's configuration.  
- - - - -  
Select the appropriate number [1-2] then [enter] (press 'c' to  
cancel): 2
```

If the process is successful, you'll get the message like below.

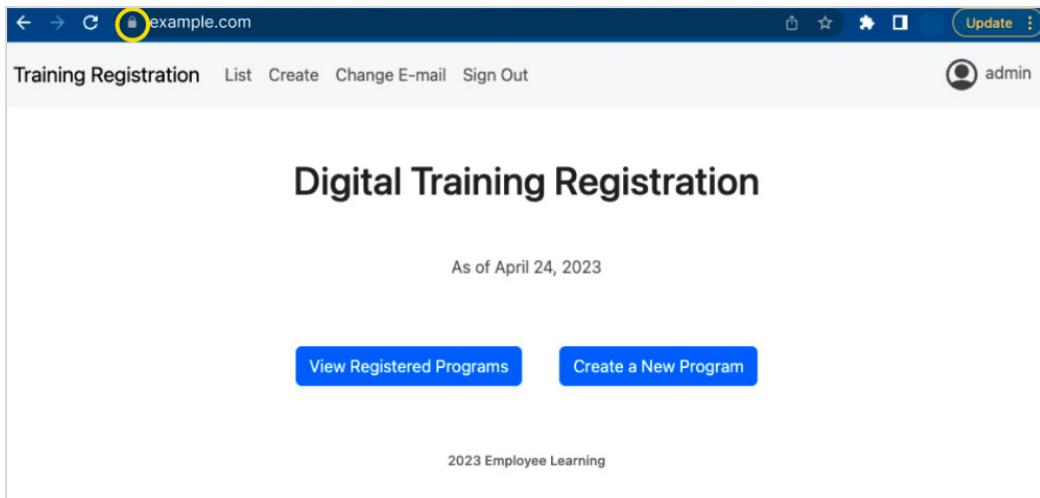
Command Line - RESPONSE

```
- - - - -  
Congratulations! You have successfully enabled https://employee-  
learning.site  
You should test your configuration at:  
https://www.ssllabs.com/ssltest/analyze.html?d=employee-  
learning.site  
- - - - -  
IMPORTANT NOTES:  
- Congratulations! Your certificate and chain have been saved at:  
/etc/letsencrypt/live/employee-learning.site/fullchain.pem  
Your key file has been saved at:  
/etc/letsencrypt/live/employee-learning.site/privkey.pem  
Your cert will expire on 2023-07-22. To obtain a new or tweaked  
version of this certificate in the future, simply run certbot  
again  
with the "certonly" option. To non-interactively renew *all* of  
your certificates, run "certbot renew"  
- If you like Certbot, please consider supporting our work by:  
Donating to ISRG / Let's Encrypt: https://letsencrypt.org/donate  
Donating to EFF: https://eff.org/donate-le
```

Check the result and test the configuration

Check your site

Now you should be able to access your website with HTTPS protocol. Go to your URL that starts with **https**.



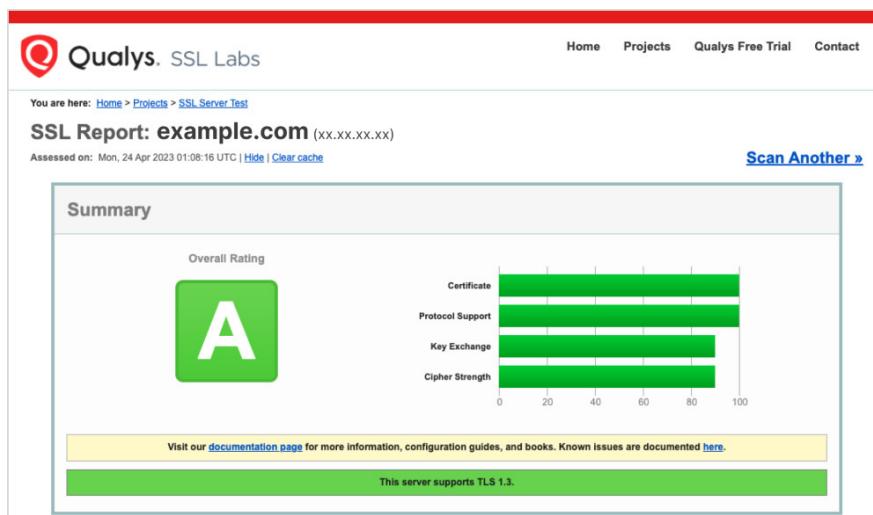
The screenshot shows a web browser window with the URL 'example.com' in the address bar. The page title is 'Digital Training Registration'. Below the title, it says 'As of April 24, 2023'. There are two blue buttons: 'View Registered Programs' and 'Create a New Program'. At the bottom left, it says '2023 Employee Learning'. The top navigation bar includes links for 'Training Registration', 'List', 'Create', 'Change E-mail', 'Sign Out', and a user profile for 'admin'.

Test the configuration

You can also test the configuration at the URL in the success message like

'<https://www.ssllabs.com/ssltest/analyze...>'.

It may take time to get the report. When the website is running properly in a secure condition, you'll see a report like the one below.



Check Nginx configuration

Certbot adds new settings in the Nginx configuration file for the project.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | cat /etc/nginx/sites-available/project_d
```

You can see that the additional settings are made by Certbot.

Command Line - RESPONSE

```
:  
listen 443 ssl; # managed by Certbot  
ssl_certificate /etc/letsencrypt/live/your_domain/fullchain.pem;  
# managed by Certbot  
ssl_certificate_key  
/etc/letsencrypt/live/your_domain/privkey.pem; # managed by  
Certbot  
include /etc/letsencrypt/options-ssl-nginx.conf; # managed by  
Certbot  
ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by  
Certbot  
}  
server {  
if ($host = your_domain) {  
return 301 https://$host$request_uri;  
} # managed by Certbot  
listen 80;  
server_name xx.xx.xx.xx your_domain;  
return 404; # managed by Certbot
```

Test automatic renewal

According to the Certbot instructions, the Certbot packages on your system come with a **cron** job or **systemd timer** that will renew your certificates automatically before they expire.

You can test automatic renewal by running the command below.

Command Line - INPUT

```
ubuntu@ip-xx:~$ | sudo certbot renew --dry-run
```

If the dry-run is successful, you'll see a success message like the one below.

Command Line - RESPONSE

```
:
```

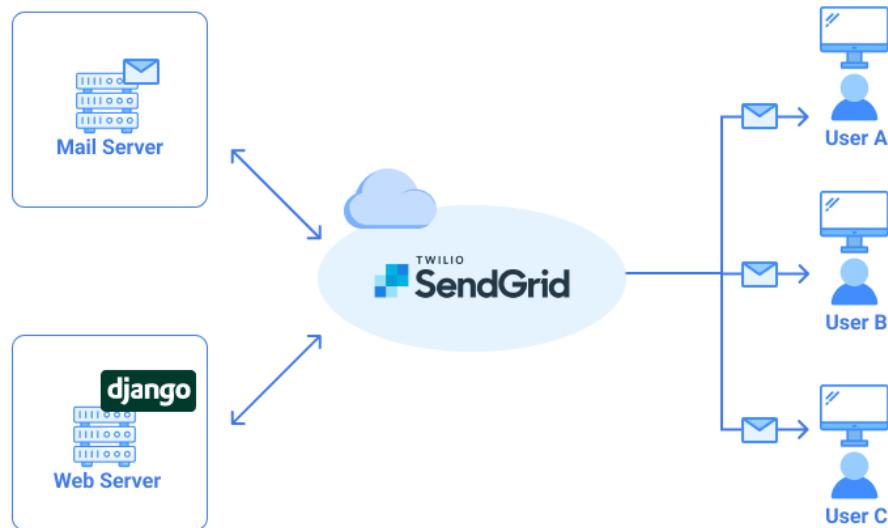
```
- - - - -
** DRY RUN: simulating 'certbot renew' close to cert expiry
** (The test certificates below have not been saved.)

Congratulations, all renewals succeeded. The following certs have
been renewed:
/etc/letsencrypt/live/your_domain/fullchain.pem (success)
** DRY RUN: simulating 'certbot renew' close to cert expiry
** (The test certificates above have not been saved.)
- - -
:
```

For more details, check certbot instructions.

[**Certbot Instructions**](#)

Email Setting – SendGrid



In the production stage, directly using your own Gmail or another email address may not be appropriate from a security and service credibility point of view. Also, when your service becomes large, your sent-email box can be full of machine-generated emails. There are third-party services that handle the machine-generated emails on behalf of your email server, such as **SendGrid**, **Amazon SES**, or **Mailtrap**. In our case example, we use SendGrid.

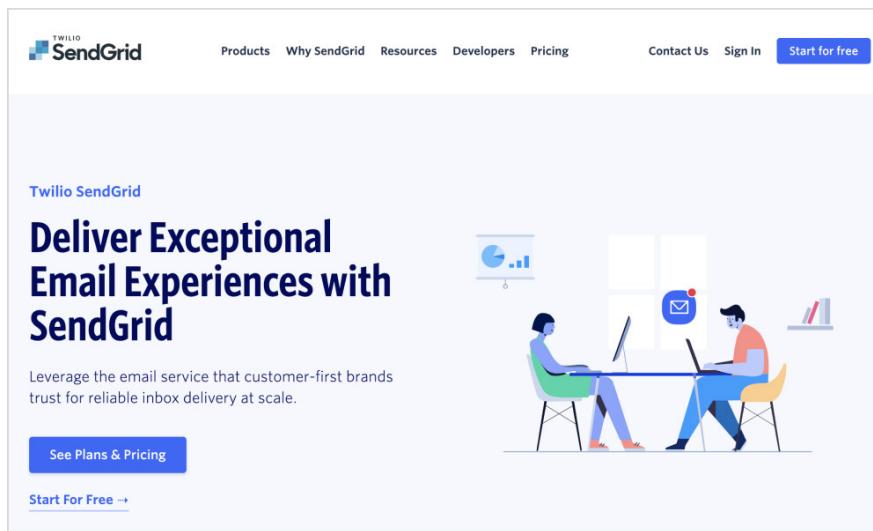
There are four key steps for SendGrid setup for Django.

1. [Sign up on SendGrid](#)
2. [Create an API Key](#)
3. [Create a sender](#)
4. [Install django-sendgrid and update the Django settings](#)

If you don't want to have the SendGrid brand name in the verification email sent by your Django app, you'll need to manage the sender authentication process on top of the four steps above.

Sign up SendGrid

Go to the SendGrid website and click on **Start for free**.



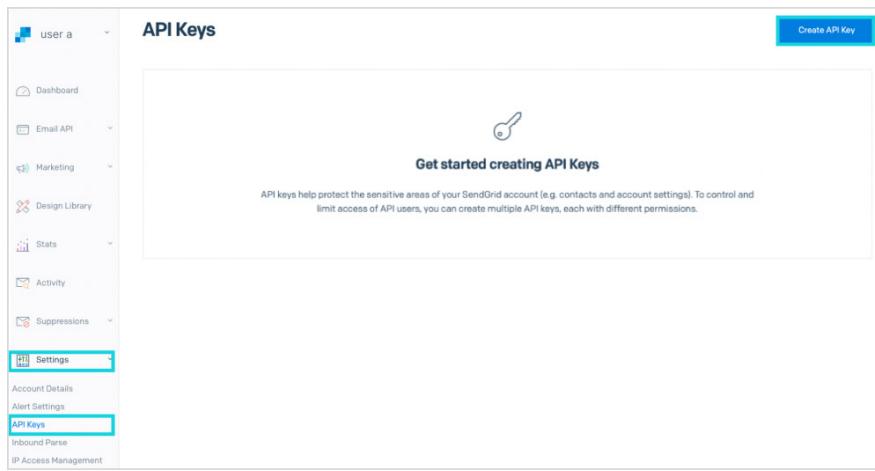
Type your email and password.

A screenshot of the "Let's Get Started" sign-up form on the SendGrid website. The form includes fields for "Email Address" (with a checked checkbox for "Use email address as username"), "Password" (with a note about needing at least 16 characters), and a reCAPTCHA verification box. There's also a checkbox for accepting the "Terms of Service" and reading the "Privacy Notice". A "Create Account" button is at the bottom. To the right of the form, there's a callout box titled "Try it out!" containing a list of features: "Send email for free.", "Automated drip campaigns", "Password resets", "Newsletters", "Receipts", "Delivery notifications and updates", and "Promoted emails". An illustration of a yellow paper airplane flying through a dashed green path is part of the design.

You also need to fill in your full name and some additional information to create an account.

Create an API Key

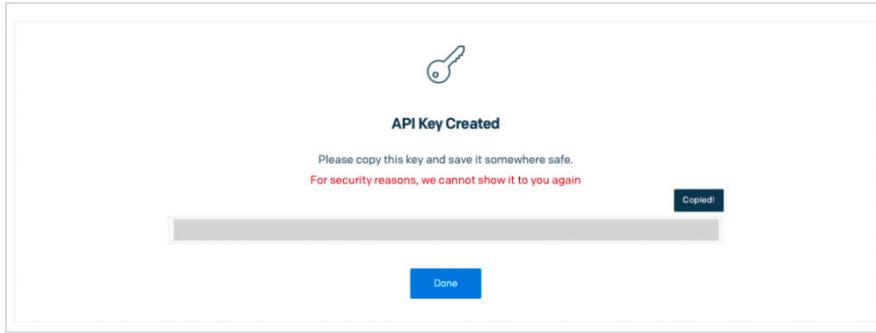
After you create an account, you need to create an API key that will be used in the Django setting later. Go to the **API Key** page under **Settings** in the left menu. Click on the **Create API Key** button.



Select **Full Access** unless you want to customize the setting.

A screenshot of the "Create API Key" dialog box. It has fields for "API Key Name" and "API Key Permissions". Under "API Key Permissions", there are three options: "Full Access" (selected), "Restricted Access", and "Billing Access". The "Full Access" option is described as allowing the API key to access all endpoints for the account, excluding billing and Email Address Validation. The "Create & View" button is at the bottom right.

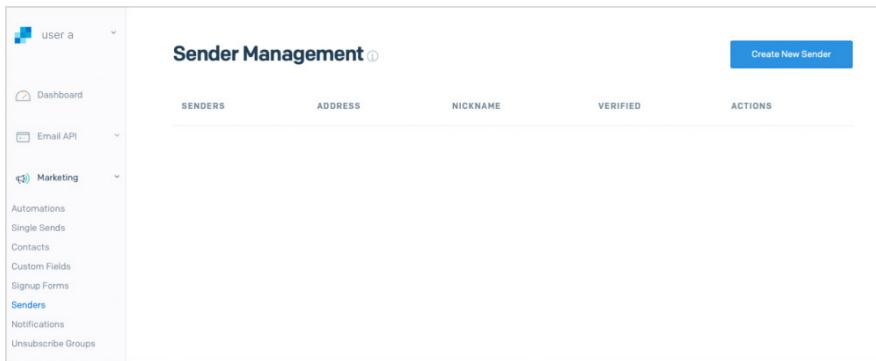
As a new API Key is shown only once, you need to make sure you copy and save it somewhere.



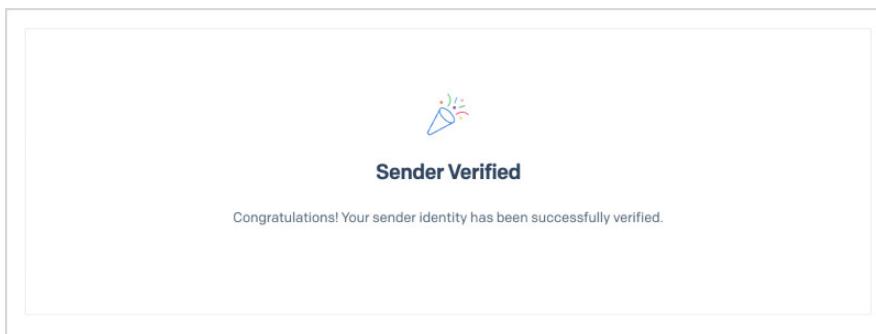
Create a sender

You need to register an email address that you want to use in the Django app.

Go to **Senders** under the **Marketing** dropdown list. Click on the **Create New Sender** button on the top right.



Add your sender information and save it. A verification email will be sent to the registered email address. Once you verify your email address, you'll get a confirmation message.



And, you'll see that your email has been registered.

Install django-sendgrid and update the Django settings

Once you get an API key and register a sender, you need to install the Django SendGrid library and update Django settings.

Install django-sendgrid

Add **django-sendgrid-v5** in the requirements file.

requirements.txt

```
Django==4.1.7
django-crispy-forms
crispy-bootstrap5
django-allauth
django-environ
gunicorn
psycopg2-binary
django-sendgrid-v5
```

and run the pip command.

Command Line - INPUT

```
(d_env) ubuntu@ip-xx-:~/project_d$ | pip install -r
requirements.txt
```

Update the Django settings file and .env file

As we won't use Gmail in the production stage, comment out or delete the settings.

Add the four settings for SendGrid.

config/production.py

```
:  
STATIC_ROOT = '/usr/share/nginx/html/static'  
  
# SendGrid  
EMAIL_BACKEND = 'sendgrid_backend.SendgridBackend'  
SENDGRID_API_KEY = env('SENDGRID_API_KEY')  
DEFAULT_FROM_EMAIL = env('DEFAULT_FROM_EMAIL')  
SENDGRID_SANDBOX_MODE_IN_DEBUG = False  
SENDGRID_TRACK_CLICKS_PLAIN = False
```

If `SENDGRID_SANDBOX_MODE_IN_DEBUG` is `True`, actual emails won't be delivered.

`SENDGRID_TRACK_CLICKS_PLAIN` is for URL tracking. If this setting is `True`, you'll see a long URL in the verification email. In our case, change it to `False`.

Actual information for the API key and default email should be saved in the `.env` file.

`.env`

```
DEFAULT_FROM_EMAIL=info@example.com  
SENDGRID_API_KEY=XXXXXXXXXXXXXXXXXXXXXX
```

For more details, please check the official document.

[django-sendgrid-v5](#)

Check if the verification email is working in the production environment

Restart the application server and check the status.

Command Line - INPUT

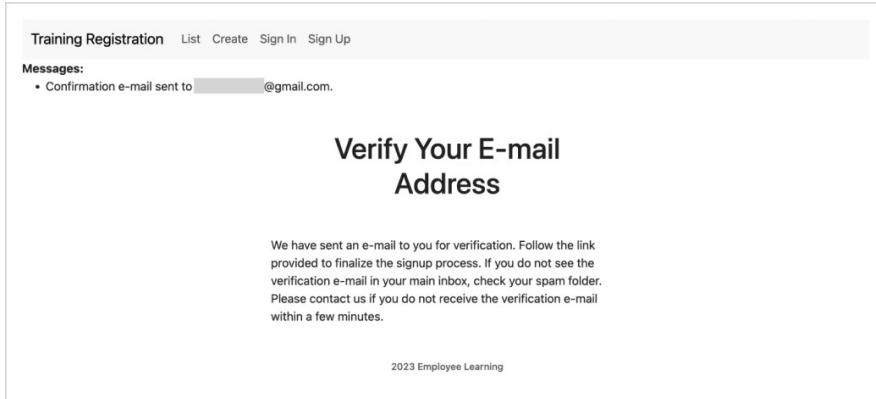
```
(d_env) ubuntu@ip-xx:~/project_d$ | sudo systemctl restart  
project_d  
(d_env) ubuntu@ip-xx:~/project_d$ | systemctl status project_d
```

Command Line - RESPONSE

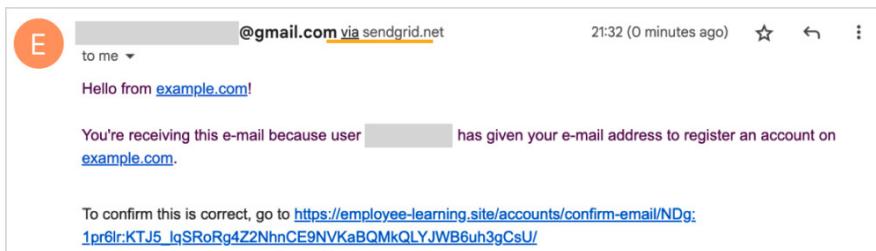
- `project_d.service` - gunicorn daemon
Loaded: loaded (/etc/systemd/system/project_d.service; disabled;
vendor preset: enabled)
Active: `active (running)` since Mon 2023-04-24 13:30:20 UTC; 1s
ago

TriggeredBy: • project_d.socket

If the app is running properly, try to sign up. You'll see that the verification email has been successfully sent.



And you'll receive an email like the one below. You can see that the email was sent via **sendgrid.net**.



Sender Authentication Setup

If you don't want to have the SendGrid name in the verification email sent by SendGrid, you need to go through the sender authentication process.

You can set it up through **Sender Authentication** under the **Settings** dropdown menu.

You can choose two approaches: **Domain Authentication** and **Single Sender Verification**.

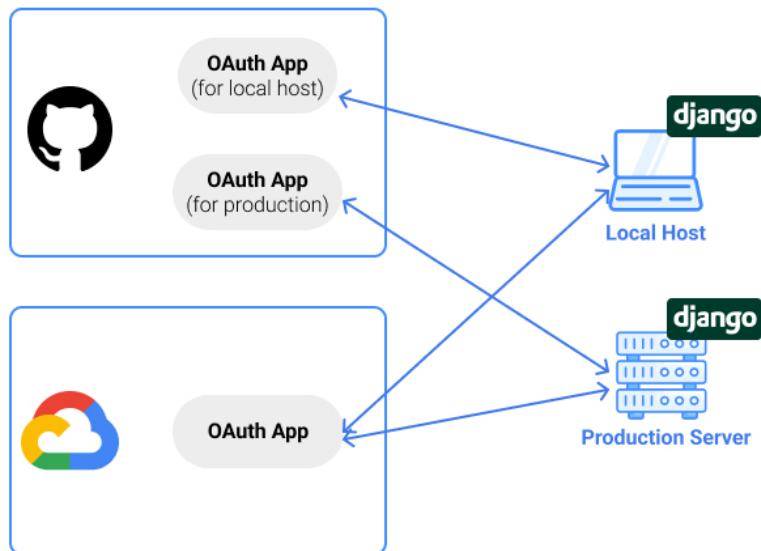
The screenshot shows the Mailjet interface for 'user a'. The left sidebar includes links for Dashboard, Email API, Marketing, Design Library, Stats, Activity, Suppressions, Settings (with sub-links for Account Details, Alert Settings, API Keys, Inbound Parse, IP Access Management, IP Addresses, Mail Settings, and Sender Authentication), and Help.

The main content area is titled 'Sender Authentication' and contains two sections:

- Sender Identity**: A section explaining that the type of Sender Identity can impact deliverability and inbox placement. It links to 'Learn more about the differences between Domain Authentication and Single Sender Verification.'
- Domain Authentication RECOMMENDED**: A section encouraging users to prove domain ownership to improve deliverability. It links to 'Learn more'.
- Single Sender Verification**: A section for verifying ownership of a single email address. It links to 'Learn more'.

If your emails are not reaching users properly, authenticating the senders may improve the situation.

Social Login for Production



As you have set social login only for localhost in the previous chapter, you need to adjust the settings (e.g., home page URL and the callback URL) for production using the domain for your web app. The steps are similar to the ones explained in the previous chapter.

[Django Allauth \(5\) – Social Login with GitHub](#)

[Django Allauth \(6\) – Social Login with Google](#)

As you are using a new database (*PostgreSQL*) and not transferring the data stored in the local database, you can make the production environment social login settings from scratch.

[**GitHub Social Login**](#)

Three steps were explained in the previous chapter. We'll explain the key differences for each step on this lesson.

1. Register a new OAuth app on the GitHub website

- Create a new app with a different name (e.g., Employee Learning (Production))
- For the home page URL and the callback URL, use the domain registered for the app instead of *localhost*

Register a new OAuth application

Application name *
Employee Learning (Production)
Something users will recognize and trust.

Homepage URL *
your-domain
The full URL to your application homepage.

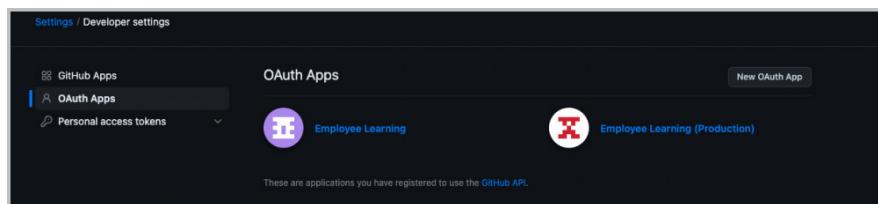
Application description
Application description is optional
This is displayed to all users of your application.

Authorization callback URL *
your-domain/accounts/github/login/callback/
Your application's callback URL. Read our [OAuth documentation](#) for more information.

Enable Device Flow
Allow this OAuth App to authorize users via the Device Flow.
Read the [Device Flow documentation](#) for more information.

Register application **Cancel**

Once the registration is done, there are two **OAuth Apps** like shown below.



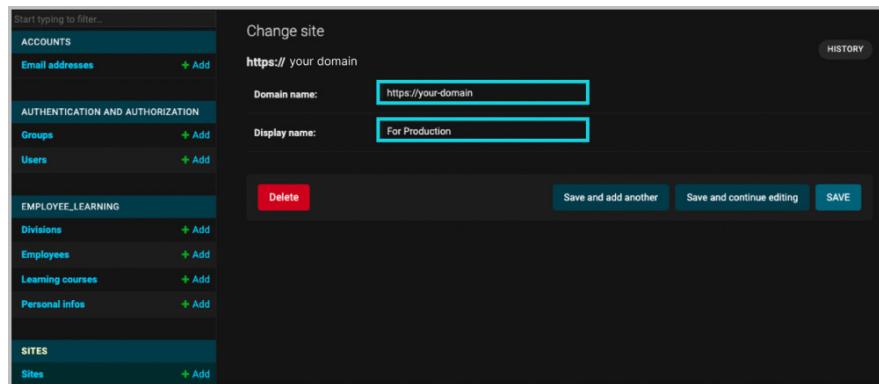
2. Edit settings.py

No need to edit the settings file unless you are using a different `SITE_ID`

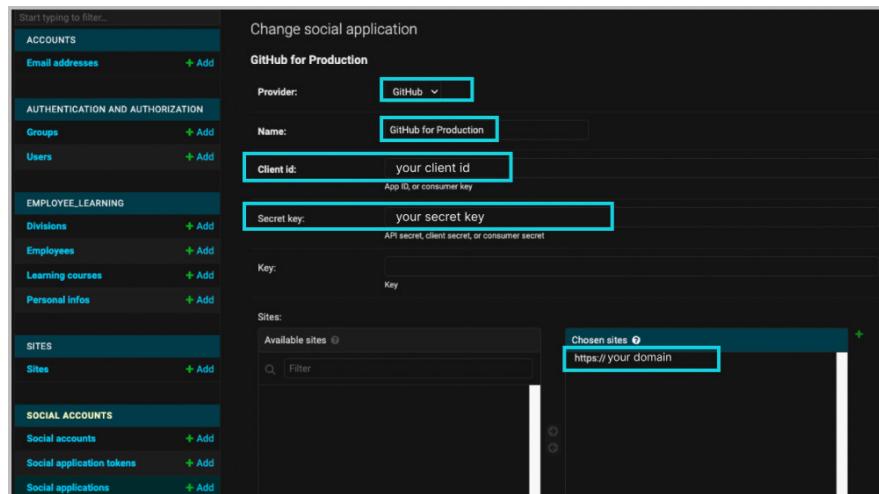
Note: `SITE_ID` is the Django ID for the site table in the Django admin.

3. Register in Django Admin

- **On the Sites page:** Update the `example.com` (`SITE_ID=1`) to your domain and new display name. If you add other site data (instead of updating `example.com`), you must change `SITE_ID` in the settings file. The updated display name will be shown in the verification email.

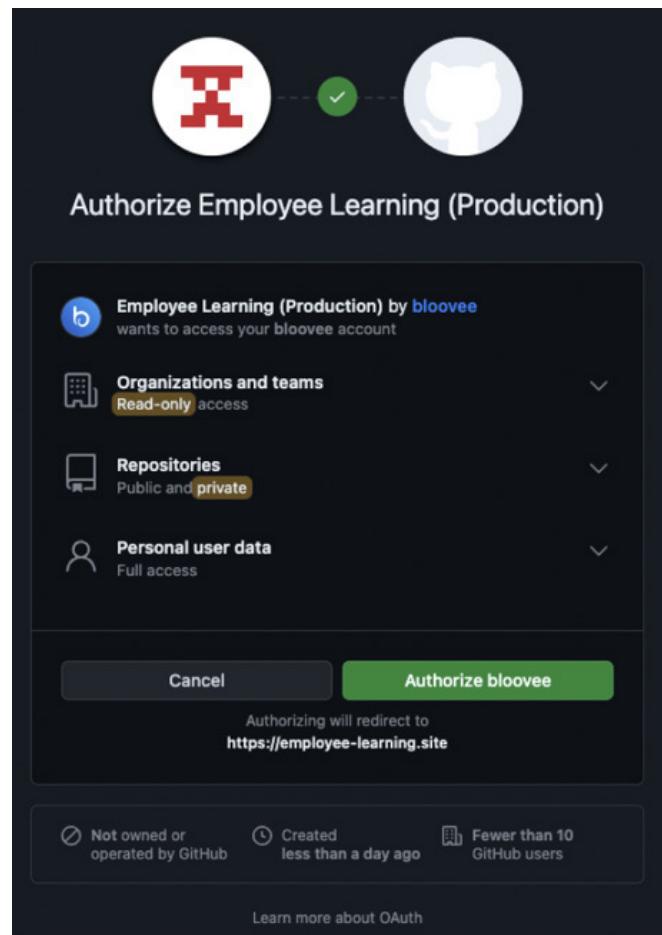


- **On the Social applications page:** Use the new **Client ID** and **Secret Key** from the new OAuth app. Add the site with your domain name.

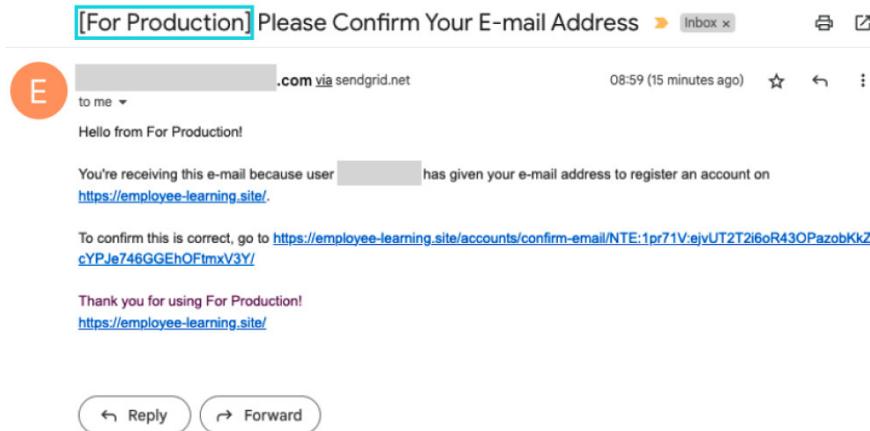


4. Check the results

Click on the GitHub icon to test if the GitHub social login is working. If all settings are correctly done, you'll be directed to the GitHub site.



Receives a verification email with the new site name.



Google Social Login

Adding a production site to the Google social login setting is easier. Unlike GitHub, you can use the same **Client ID** and **Secret Key** that you generated for the *localhost*.

You need to do the following:

1. Update credentials (the new domain address and redirect the URL) to the GCP platform
2. Add the new social application data to the Django admin page.

1. Update Credentials on the GCP platform

Google Cloud Employee Learning

API APIs and services Client ID for Web application DELETE

Enabled APIs and services

Library

Credentials

OAuth consent screen

Page usage agreements

Name * Employee Learning

The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

The domains of the URIs you add below will be automatically added to your [OAuth consent screen](#) as [authorised domains](#).

Authorised JavaScript origins

For use with requests from a browser

URIs 1 * http://localhost:8000

URIs 2 * https:// your domain

+ ADD URI

Authorised redirect URIs

For use with requests from a web server

URIs 1 * http://localhost:8000/accounts/google/login/callback/

URIs 2 * https:// your domain /accounts/google/login/callback/

+ ADD URI

Note: It may take five minutes to a few hours for settings to take effect

SAVE CANCEL

2. Add a new social application on the Django admin platform

Django administration

Home : Social Accounts : Social applications : Add social application

Add social application

Provider: Google

Name: Google for Production

Client id: your client id

Secret key: your secret key

Key:

Sites:

Available sites	Chosen sites
https://your domain	https://your domain

Choose all Remove all

Hold down "Control", or "Command" on a Mac, to select more than one.

Save and add another | Save and continue editing | SAVE

3. Check the results

Now, you have two social applications for production.

Select social application to change		ADD SOCIAL APPLICATION +
Action:	———	Go 0 of 2 selected
<input type="checkbox"/>	NAME	PROVIDER
<input type="checkbox"/>	Google for Production	Google
<input type="checkbox"/>	GitHub for Production	GitHub
2 social applications		

Click on the Google icon. You should be able to use the Google social login feature now.

Training Registration List Create Sign In Sign Up

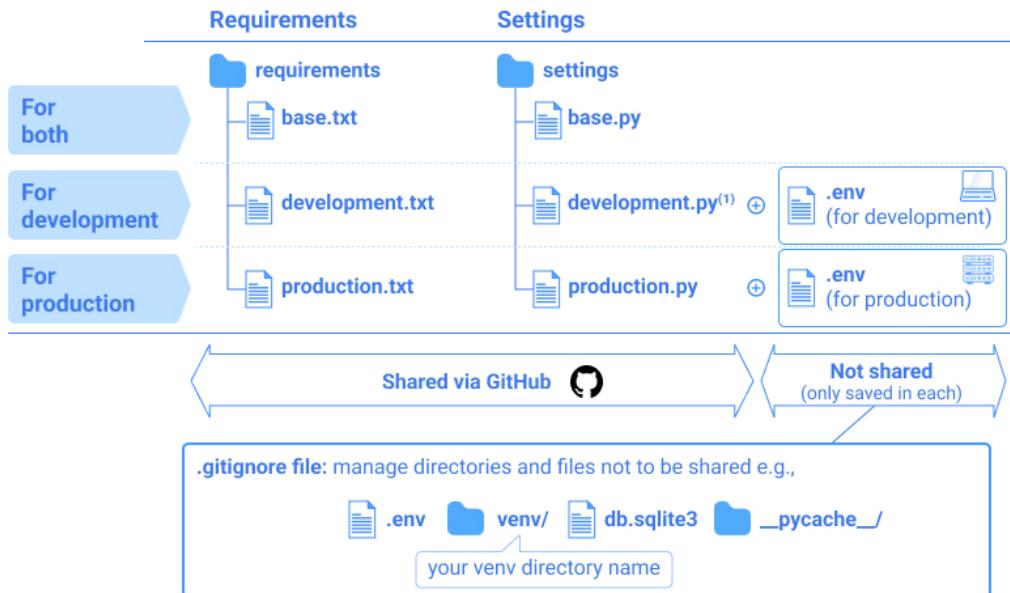
Sign In Via Google

You are about to sign in using a third party account from Google.

Continue

2023 Employee Learning

Manage Local Development and Remote Production Environment



Note (1): you can also ignore this file instead of managing confidential data through .env file

As the last topic of this course, we'll explain how to efficiently manage the **local development environment** and the **remote production environment**. When you launch your new web service and work with other developers, you'll need to manage both environments simultaneously. As explained earlier, there are three key points in managing the differences between the deployment and production environments.

1. Do not share irrelevant files (.gitignore)
2. Do not install irrelevant dependencies (requirements.txt)
3. Do not set irrelevant configurations (settings.py)

This sounds easy, but it is troublesome in practice. Here, we'll summarize the key approaches using our case.

Requirements

We haven't yet explained how to differentiate the requirements files for development and production. We can use the same approach as the one used for settings files. Create the *requirements* directory and add three files to manage requirements: *base.txt*, *development.txt*, and *production.txt*. Here are the examples of the three files used in our app example.

base.txt

Rename the original *requirements.txt*, and keep the requirements that are commonly used for both environments.

`requirements/base.txt`

```
Django==4.1.7
django-crispy-forms
crispy-bootstrap5
django-allauth
django-environ
```

development.txt

In our case, we are not using any specific library for development. Just get the list from *base.txt* by writing `-r base.txt`.

`requirements/development.txt`

```
-r base.txt
```

production.txt

Use `-r base.txt` to get the list from *base.txt*, and add production-specific requirements — just like you did for *development.txt*.

- gunicorn : application server
- psycopg2-binary : PostgreSQL adapter (PostgreSQL itself is directly installed on Linux)

- django-sendgrid-v5 : for SendGrid

requirements/production.txt

```
-r base.txt  
gunicorn  
psycopg2-binary  
django-sendgrid-v5
```

In practice, we usually use more libraries. This approach will be helpful in organizing your requirements for different environments.

The pip command

When you run the `pip` command, you need to adjust the file path. For example, when you run the command in the local development environment, run the command like shown below.

Command Line - INPUT

```
(d_env) ubuntu@ip-xx:~/project_d$ | pip install -r  
requirements/development.txt
```

When you run the command in the remote production environment, run the command below.

Command Line - INPUT

```
(d_env) ubuntu@ip-xx:~/project_d$ | pip install -r  
requirements/production.txt
```

Settings

We have already explained how to structure the settings files with an example for production settings. Here, we'll summarize the final version of both files.

development.py

Bring key settings specific to the local development environment. If you have the data that you want to manage confidentially, use the `.env` file. You can also ignore this `development.py` file using the `.gitignore` file instead of creating the `.env` file for development.

settings/development.py

```

from .base import *
import environ

env = environ.Env()
env.read_env('.env')

SECRET_KEY = env('SECRET_KEY')

DEBUG = True

ALLOWED_HOSTS = env.list('ALLOWED_HOSTS')

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
# Console email
#EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'

# Gmail
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = env('EMAIL_HOST_USER')
EMAIL_HOST_PASSWORD = env('EMAIL_HOST_PASSWORD')
EMAIL_PORT = 465
EMAIL_USE_TLS = False
EMAIL_USE_SSL = True
DEFAULT_FROM_EMAIL = env('DEFAULT_FROM_EMAIL')

```

`SECRET_KEY` is the same for both development and production, but we put it in each requirement file to create a consistent structure.

For `EMAIL_BACKEND`, keep two options – console and Gmail, but one setting should be commented out.

Here is an example of an `.env` file for development. `ALLOWED_HOSTS` can be directly written in the `development.py` file. This is the case when you are using your **private IP address** for a mobile UI design, which is explained on this topic page [Check Developing App UI on Mobile Device](#).

`.env(for development)`

```

SECRET_KEY=your_secret_key
ALLOWED_HOSTS=localhost,your_private_IP_address

# Gmail
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'App Password'
DEFAULT_FROM_EMAIL = 'your_account@gmail.com'

```

production.py

Here is the final version of the *productino.py* example. Put confidential information in the *.env* file, just like you did for *development.py*.

settings/production.py

```
from .base import *
import environ

env = environ.Env()
env.read_env('.env')
SECRET_KEY = env('SECRET_KEY')

DEBUG = False

ALLOWED_HOSTS = env.list('ALLOWED_HOSTS')

DATABASES = {
    'default': {
        'ENGINE': env('DB_ENGINE'),
        'NAME': env('DB_NAME'),
        'USER': env('DB_USER'),
        'PASSWORD': env('DB_PASSWORD'),
        'HOST': env('DB_HOST'),
        'PORT': env('DB_PORT'),
    }
}

STATIC_ROOT = '/usr/share/nginx/html/static'

# SendGrid
EMAIL_BACKEND = 'sendgrid_backend.SendgridBackend'
SENDGRID_API_KEY = env('SENDGRID_API_KEY')
DEFAULT_FROM_EMAIL = env('DEFAULT_FROM_EMAIL')
SENDGRID_SANDBOX_MODE_IN_DEBUG = False
SENDGRID_TRACK_CLICKS_PLAIN = False
```

In the *.env* file, we defined the following settings:

- SECRET_KEY
- ALLOWED_HOSTS
- DB settings
- SENDGRID settings

.env(for production)

```
SECRET_KEY=your_secret_key
ALLOWED_HOSTS=your_public_IP_address,your_site_domain
```

```
DB_ENGINE=django.db.backends.postgresql
DB_NAME=project_d
DB_USER=project_d_user
DB_PASSWORD=project_d_pass
DB_HOST=localhost
DB_PORT=5432

DEFAULT_FROM_EMAIL=your_email
SENDGRID_API_KEY=your_API_key
```

Managing the `.gitignore` file

Managing the `.gitignore` file is very important in practice. If you forget to list files that should be ignored and push your code to a Git repository, reversing this action requires a lot of effort. Thus, you need to be very careful to list all the key files that should not be shared before pushing your code.

You can use [gitignore.io](#) to create your `.gitignore` file, but you need to make sure that you add your virtual environment directory to the list as you define the directory name. As the development and production environments are different, sharing this directory can create several problems.

Frequently used commands

As key commands for the local environment and the production environment are slightly different (e.g., file path), it is beneficial to list them up so that you can copy and paste them when you run the command.

Local development environment

Here is a list of frequently used commands for the local environment that uses our example project (you need to update the commands based on your file name or path setting).

```
project_d % | rm -r d_env
project_d % | python3 -m venv d_env
project_d % | source d_env/bin/activate
```

```
(d_env) project_d % | python -m pip install --upgrade pip
```

```
(d_env) project_d % | pip install -r requirements/development.txt
(d_env) project_d % | python manage.py makemigrations --settings
config.settings.development
(d_env) project_d % | python manage.py migrate --settings
config.settings.development
(d_env) project_d % | python manage.py createsuperuser --settings
config.settings.development
(d_env) project_d % | python manage.py runserver --settings
config.settings.development
```

Production environment

Here is a list of frequently used commands for the production environment. The key differences with the list above are the requirement file path and the way of running the Django app. As you cannot use the `runserver` command in the production environment, you need to use `systemctl` commands.

```
ubuntu@ip-xx:~/project_d$ | rm -r d_env
ubuntu@ip-xx:~/project_d$ | python3 -m venv d_env
ubuntu@ip-xx:~/project_d$ | source d_env/bin/activate
```

```
(d_env) ubuntu@ip-xx:~/project_d$ | python -m pip install --
upgrade pip pip install -r requirements/production.txt
(d_env) ubuntu@ip-xx:~/project_d$ | python manage.py
makemigrations
(d_env) ubuntu@ip-xx:~/project_d$ | python manage.py migrate
(d_env) ubuntu@ip-xx:~/project_d$ | python manage.py
createsuperuser
(d_env) ubuntu@ip-xx:~/project_d$ | python manage.py
collectstatic
(d_env) ubuntu@ip-xx:~/project_d$ | sudo systemctl restart
project_d
(d_env) ubuntu@ip-xx:~/project_d$ | systemctl status project_d
```

About D-Libro



<https://d-libro.com/>

D-Libro is a learning platform and eBook library specialized in digital skills, including:

- Programming and coding
- Web and application design
- Cloud-based infrastructure management
- Digital Marketing

This book is edited based on the **Django Introduction** course on the D-Libro platform.



<https://d-libro.com/course/django-introduction/>

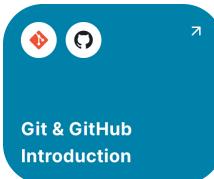
There are several courses available for beginners.

HTML&CSS Introduction



<https://d-libro.com/course/html-css-introduction/>

Git & GitHub Introduction



<https://d-libro.com/course/git-github-introduction/>

Linux Introduction



<https://d-libro.com/course/linux-introduction/>

SEO Tutorial for Beginners



<https://d-libro.com/course/seo-tutorial-for-beginners/>