# Mastering Django: Projects for Beginners

A Beginner's Journey to Django Mastery: Projects and Beyond

**Aditya Dhandi**

## Also by Aditya Dhandi

- Python Course - Learn Python From Scratch : 

# Index

# Chapter 1: Introduction

Welcome to the immersive world of “Mastering Django: Projects for Beginners”—a book that places projects at the core of your journey into web development using the powerful Django web framework.

This book is meticulously crafted to be hands-on, ensuring your comprehension through an abundance of examples and exercises in every chapter. Whether you're delving into the content sequentially or employing it as a targeted reference, consider this book your steadfast companion on your Django expedition.

Whether you're on the brink of creating a personal project, launching a startup, or simply navigating the expansive realm of web development, Django stands as your ideal companion. With “Projects for Beginners” serving as your navigator, you'll swiftly evolve into a developer capable of crafting impactful web applications that leave a lasting imprint on the digital landscape.

Brace yourself to convert your aspirations in web development into tangible achievements as we commence our exciting Django adventure together!

Just as our approach mirrors the successful structure of “Projects for Beginners”, where you construct progressively intricate web applications, from a humble "Hello, World" app to a sophisticated Job Listing application with advanced features—forms, user accounts, custom user models, email integration, foreign keys, authorization, permissions, and beyond—you can expect a similar journey in “Mastering Django: Projects for Beginners”.

By the final pages of this book, you'll emerge with the confidence to create your own Django projects from scratch, armed with the latest best practices in web development. Django, a free, open-source web framework written in Python and embraced by millions of developers worldwide, seamlessly caters to both beginners and advanced programmers.

While robust enough for the largest websites globally—Instagram, Pinterest, Bitbucket, Disqus—Django remains flexible, making it an excellent choice for early-stage startups and personal projects.

This book is a living document, regularly updated to align with the latest versions of Django(4.0) and Python (3.11x). We adopt Pipenv, the officially recommended package manager by Python.org, for handling Python packages and virtual environments.

Throughout our exploration, we adhere to modern best practices from the Django, Python, and web development communities, placing a strong emphasis on thorough testing—a fundamental aspect of building robust and scalable applications. So, let's embark on this journey, leveraging the collective wisdom and experiences of the web development community to master the art of Django.

# What You'll Learn

In the upcoming sections, I'll guide you through the Django framework in a systematic manner. We'll initiate by configuring your development environment and initiating your inaugural Django project. Following this, we'll delve into subjects such as models and databases, views and templates, and the fundamental principles required to construct a web application from the ground up.

We'll explore aspects like forms, authentication, and user management, ensuring the functionality of your application while prioritizing security and user-friendliness. As you advance, you'll confront more intricate subjects such as working with static and media files, employing class-based views, navigating the Django ORM, conducting tests, debugging, and executing deployment.

I'll provide you with the tools and insights essential for crafting your unique application, and we'll also investigate supplementary functionalities, including real-time capabilities.

Upon completing this resource, you'll not only possess a robust understanding of Django but also the assurance to initiate your web development endeavors. Armed with adept practices and optimization techniques, you'll be prepared to fabricate efficient, scalable, and secure web applications.

# What is Django

Django is a high-level, open-source web framework for building web applications using the Python programming language.

It follows the Model-View-Controller (MVC) architectural pattern, which is often referred to as Model-View-Template (MVT) in Django's context.

Django's primary goal is to simplify and speed up the process of developing web applications by providing a robust set of tools and conventions for common web development tasks.

Here are some key features and components of Django:

- **ORM (Object-Relational Mapping):** Django includes a powerful ORM that allows developers to define their data models using Python classes. These models are then translated into database tables, and developers can interact with the database using Python code, rather than writing SQL queries directly.

- **Admin Interface:** Django comes with an automatic admin interface, which can be customized to manage your application's data models. This is especially helpful for content management and administrative tasks.

- **URL Routing:** Django uses a URL dispatcher to map URLs to views. This allows for clean and flexible URL routing, making it easy to define how different parts of your application respond to different URL patterns.

- **Template Engine:** Django has a template engine that allows developers to define the structure and layout of HTML pages in a clean and efficient way, separating the presentation layer from the business logic.

- **Authentication and Authorization:** Django provides built-in tools for user authentication and authorization, making it straightforward to add user accounts and manage permissions.

- **Security:** Django includes various built-in security features to help developers protect their applications against common web vulnerabilities, such as cross-site scripting (XSS) and SQL injection.

- **Middleware:** Django allows you to define middleware components that can process requests and responses globally, enabling tasks like authentication, logging, and more.

- **Forms and Validation:** Django provides a forms framework that simplifies the creation and handling of HTML forms, including form validation.

- **Database Support:** Django supports multiple databases, including PostgreSQL, MySQL, SQLite, and Oracle, allowing developers to choose the database that best suits their needs.

- **REST Framework:** While not part of Django core, the Django REST framework is a popular third-party package for building robust RESTful APIs using Django.

- **Community and Ecosystem:** Django has a large and active community, which means there are many reusable packages and extensions available to help streamline development.

Django's "batteries-included" philosophy means it comes with many built-in features, which can accelerate development while maintaining best practices. It's an excellent choice for building a wide range of web applications, from simple websites to complex, data-driven platforms.

# Why Django

Why choose Django as your trusty steed on the vast and ever-evolving journey of web development? In a digital world filled with countless frameworks and tools, Django shines as a beacon of innovation, practicality, and sheer magic. Here's why Django is the sorcerer's stone of web development:

- **Pythonic Elegance:** Django is built on Python, a language known for its readability and simplicity. This means you'll spend less time deciphering cryptic code and more time crafting your web applications. With Django, you're working in a programming language that feels like poetry.

- **Batteries Included:** Django doesn't just offer a web framework; it comes fully equipped with a treasure trove of tools and utilities. From an admin panel for data management to authentication and security features, Django provides everything you need to create robust web applications without the hassle of piecing together numerous third-party libraries.

- **Rapid Development:** Django follows the "Don't Repeat Yourself" (DRY) principle. It automates many common development tasks, allowing you to focus on what makes your application unique. This accelerated development process means you can bring your ideas to life faster than ever before.

- **Scalability and Flexibility:** Django is not just for small projects. It scales seamlessly, making it suitable for both startups and large enterprises. Its modular architecture allows you to use only the components you need, providing flexibility to tailor your development stack.

- **Community and Documentation:** Django boasts a vibrant and supportive community of developers, readily available to answer questions and provide guidance. Its extensive documentation serves as a treasure map, guiding you through every corner of the framework.

- **Security Enchanted:** Security is a paramount concern in the digital realm, and Django takes it seriously. The framework comes equipped with built-in defenses against common web vulnerabilities, such as SQL injection and cross-site scripting (XSS). This means you can focus on building your application while Django guards it against malicious sorcery.

- **Battle-Tested and Trusted:** Django has been used to power countless websites, from small personal blogs to major platforms like **Instagram** and **Pinterest**. Its track record speaks to its reliability and scalability.

- **Versatile Ecosystem:** Django's ecosystem extends beyond web development. It offers tools and libraries for tasks like creating RESTful APIs with the Django REST framework, building real-time applications with Django Channels, and integrating with other Python packages for data analysis, machine learning, and more.

- **Open Source Magic:** Django is open source, meaning it's free to use and has a community that continually contributes to its improvement. This open nature encourages innovation and ensures that Django remains relevant and up-to-date.

- **Future-Proof:** Django evolves with the ever-changing landscape of web development. Its maintainers and community work tirelessly to keep it in sync with modern practices and technologies, making it a reliable choice for the long haul.

In summary, Django isn't just a web framework; it's an enchanting toolkit for web sorcerers. With its Pythonic elegance, comprehensive features, and strong community support, Django empowers developers to create web applications that are not only functional but truly magical. So, embrace the Django magic, and let your web development journey begin!

# Django Vs Flask

Django and Flask are both popular Python web frameworks, but they have different philosophies, use cases, and levels of complexity. The choice between Django and Flask depends on your project requirements and your development preferences. Here's a comparison of Django and Flask:

## Django:

**Philosophy:** Django follows the "batteries-included" philosophy, which means it provides a comprehensive set of tools and features for web development out of the box. It includes an ORM, an admin interface, authentication, and more.

**Complexity:** Django is a high-level framework that enforces a specific project structure and follows the Model-View-Controller (MVC) architectural pattern, which is often referred to as Model-View-Template (MVT) in Django's context. This makes it suitable for complex, large-scale web applications where a lot of built-in functionality is required.

**Built-in Features:** Django includes many built-in features like an admin interface, user authentication, database migrations, and an ORM. This can save time and effort for developers.

**Learning Curve**: Due to its comprehensive nature, Django can have a steeper learning curve for beginners. However, it provides clear guidelines and conventions for development, which can be beneficial in the long run.

**Community and Ecosystem:** Django has a large and active community, along with a rich ecosystem of third-party packages and extensions.

**Use Cases:** Django is often chosen for large-scale, data-driven applications, content management systems (CMS), and projects that require a lot of built-in functionality.

Here are some popular websites and platforms that use Django:

- Instagram
- Pinterest
- The Washington Post
- Eventbrite
- Disqus
- NASA
- Bitbucket
- Mozilla Support

- The Onion
- The Guardian
- Prezi
- Mahalo

## Flask:

**Philosophy:** Flask follows the "micro" framework philosophy, which means it provides the basic tools and leaves many decisions to the developer. The flask is minimalistic and lightweight.

**Complexity:** Flask is simpler and more flexible compared to Django. It doesn't impose a specific project structure or architecture, allowing developers to choose their preferred components and libraries.

**Extensibility:** Flask is highly extensible and allows developers to add only the components they need. This results in more control and flexibility but also requires more decisions on the developer's part.

**Learning Curve:** Flask has a lower learning curve, making it a good choice for beginners or projects that need to be developed quickly.

**Community and Ecosystem:** Flask has a vibrant community and a range of extensions, but its ecosystem is not as extensive as Django's.

**Use Cases:** Flask is often chosen for smaller to medium-sized projects, RESTful APIs, microservices, and projects where developers want more control over the components they use.

In summary, Django is a full-featured framework with a lot of built-in functionality, making it suitable for complex and feature-rich applications. Flask, on the other hand, is lightweight and flexible, giving developers more control over the components they use, making it a great choice for smaller projects or when simplicity and customizability are priorities. The choice between the two depends on the specific needs of your project and your development preferences.

# Why this Book

"**Mastering Django: Projects for Beginners "** serves as your indispensable companion for honing the skills of web development using Django. Whether you're new to programming or making a transition from another framework, this book is tailored to meet your educational requirements.

## Key Features:

**Structured Learning Path:** Follow a meticulously designed curriculum that seamlessly guides you from fundamental concepts to advanced Django topics, ensuring a well-paced learning experience.

**Hands-On Examples:** Immerse yourself in real-world examples and practical exercises that solidify your grasp of Django fundamentals. These hands-on activities make intricate concepts understandable and applicable.

**Comprehensive Coverage:** Delve into the complete Django ecosystem, covering the spectrum from crafting dynamic web applications to understanding databases, authentication, and deployment. Gain a comprehensive understanding of full-stack development.

**Problem-Solving Approach:** Develop the skills to address common challenges encountered by developers. Detailed explanations and solutions empower you to navigate obstacles independently.

Embark on your Django development journey confidently with "**Django Fundamentals."** This book provides you with the skills and knowledge necessary to construct robust web applications. Unleash the potential of Django and commence the creation of dynamic, scalable, and secure websites today.

# Book layout

In this book, you'll encounter various code examples, which we'll identify using the following conventions:

```
# This is Python code
print("Hello, World")
```

To keep things concise, we'll represent unchanged, pre-existing code with dots (...) when, for instance, we're making updates within a function:

```
def this_is_my_function:
    ...
    print("Completed")
```

Throughout the book, we'll also use the command line console extensively. Commands will be preceded by a **>** symbol, following the traditional Unix style:

***command prompt***

```
> echo "Hello World"
```

The resulting output of a command will be presented without the **>** symbol:

***command prompt***

```
"Hello World"
```

To simplify comprehension, we'll often display both the command and its output together:

***command prompt***

```
> echo "Hello World"
Hello World
```

For the complete source code of all examples, please refer to the official GitHub repository.

## A Brief Overview of Python

Before we dive headfirst into Django's enchanting world, it's essential to have a basic understanding of the magic wand we'll be wielding—Python. If you're already familiar with Python, consider this a quick refresher; If you're new to it, please have some understanding of Python.

### What Is Python?

Python is a versatile, high-level programming language celebrated for its simplicity and legibility. Think of Python as a versatile spellbook, packed with readable and expressive spells (code) that can accomplish a wide range of tasks. It's a language that emphasizes clean and concise code, making it an ideal choice for beginners and seasoned developers alike.

### Python's Philosophy

Python follows a set of guiding principles known as "The Zen of Python" or "PEP 20." some of these guiding principles include:

- Readability counts: Code should be easy to read and understand.
- Simple is better than complex: Python encourages simplicity in code design.
- There should be one-- and preferably only one --obvious way to do it: Python promotes a single, clear way of accomplishing tasks.

### Python's Strengths

Python's versatility extends to various domains:

- **Web Development:** Django and Flask are popular Python frameworks for building web applications.
- **Data Science:** Python is a go-to choice for data analysis, machine learning, and scientific computing, with libraries like NumPy, Pandas, and sci-kit-learn.
- **Automation:** Python's simplicity makes it perfect for scripting tasks, automating repetitive actions, and working with APIs.
- **Game Development:** Pygame allows you to create 2D games with Python.
- **Scripting:** It's often used for writing scripts to automate tasks on servers and in system administration.

### Python Syntax

Python uses a clean and minimalistic syntax. Here's an example:

```
# This is a Python comment

def greet(name):
    message = "Hello, " + name + "!"
    print(message)

greet("Alice")
```

In Python, indentation (whitespace) is significant, and code blocks are defined by colons and indentation levels.

### Data Types

Python supports various data types, including

Integers (int)
Floating-point numbers (float)
Strings (str)
Lists (list)
Dictionaries (dict)
Booleans (bool)
Tuples (tuple)
Sets (set)

### Control Structures

Python provides control structures like loops and conditionals. Here's an example:

```
for i in range(5):
    if i % 2 == 0:
        print(i,"is even")
    else:
        print(i,"is odd")
```

### Functions

You can define functions in Python to encapsulate blocks of code for reuse:

```
def add(x, y):
    return x + y

result = add(3, 5)
```

### Libraries and Modules

Python's power extends through its vast library ecosystem. You can import and use modules such as math, random, and others to extend Python's capabilities.

### Your Python Toolkit

As you embark on your Django journey, Python will be your trusty wand. You'll cast spells (write code) with it to create the magic of web applications. Don't worry if you're new to Python; we'll guide you through the incantations and reveal the secrets of Django along the way. So, let's step into Python's mystical world together.

### Python Resources

If you are new to Python, you can learn Python from the following books:

- Python Crash Course
- Think Python
- The Hitchhiker's Guide to Python

# Chapter 2: Setting up a development environment

This chapter guides on configuring your computer effectively for working on Django projects. We begin with an overview of the command line and utilize it to install the latest versions of both Django (4.0), and Python (3.11). Subsequently, we delve into topics such as virtual environments, git, and working with a text editor.
By the conclusion of this chapter, you will be well-prepared to create and modify Django projects with ease.

### The Command Line

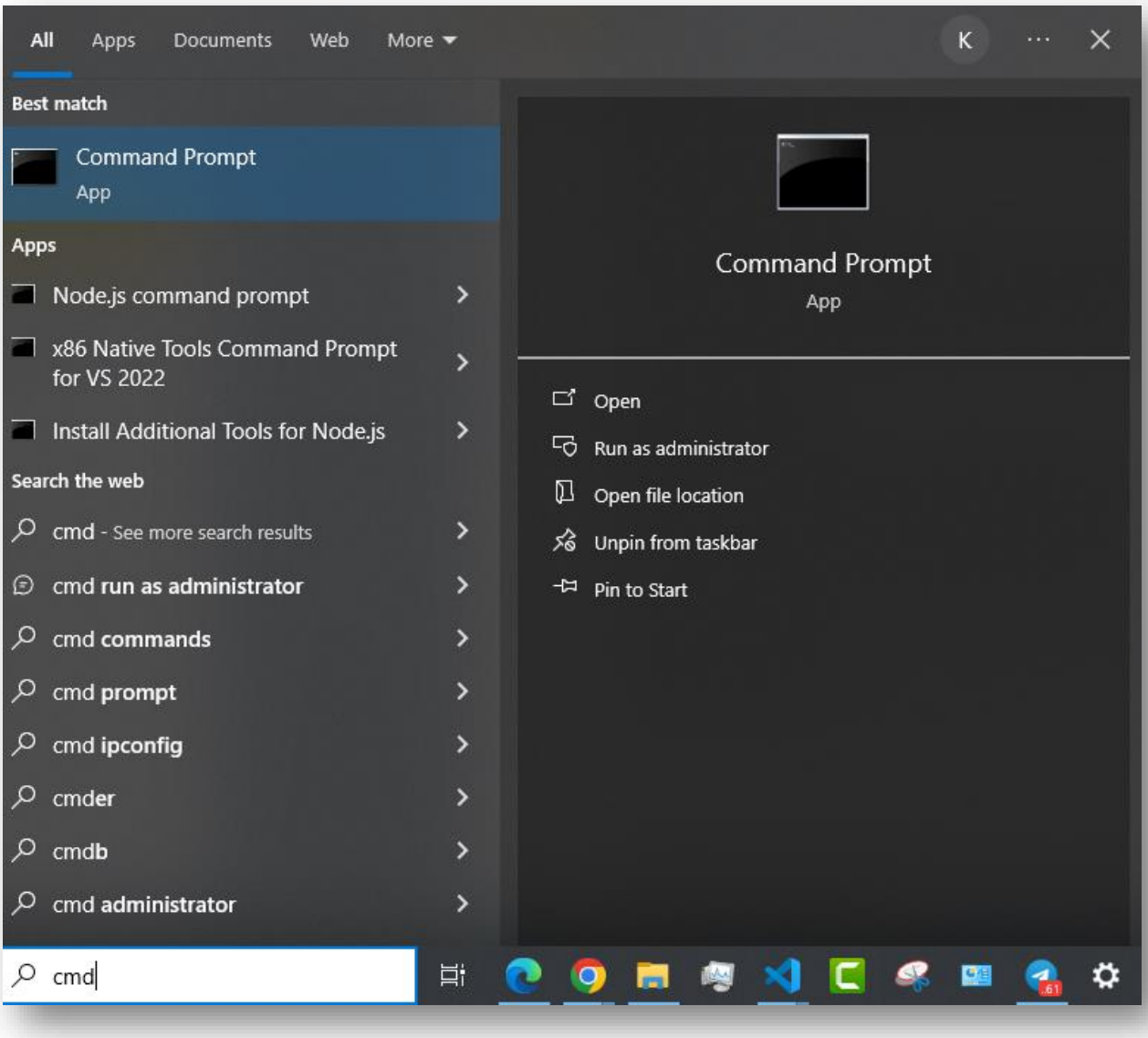
The command line is a potent, text-based interface to your computer. As developers, we will utilize it extensively throughout this guide for installing and configuring each Django project.

On a Mac, you can access the command line via a program called Terminal, located at **/Applications/Utilities.**

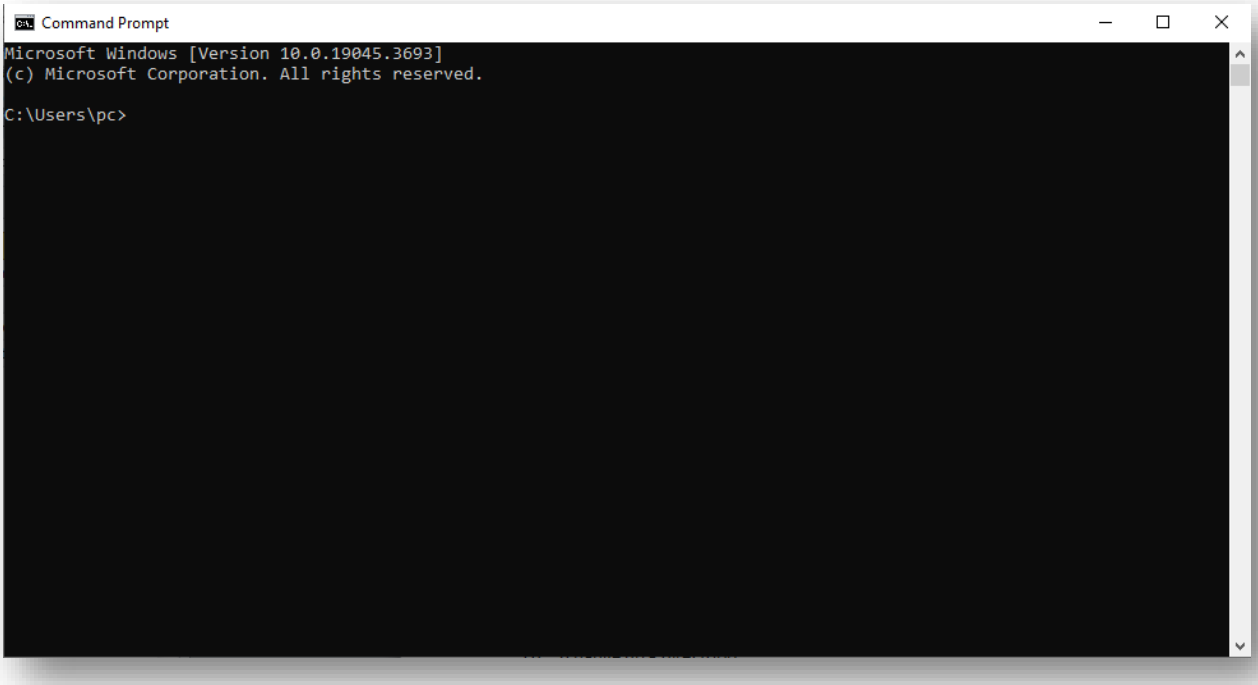
To find it, open a new Finder window, navigate to the Applications folder, scroll down to open the Utilities folder, and double-click the Terminal application.

For Windows users, there is an integrated command line program called Command Prompt. From here on, when we mention the "command line," we mean opening a new console on your computer using Terminal.

To get the command line, go to the search box of your system and type **cmd**. You will see something like this:




Just click on Command Prompt to open the command line.




While there are numerous commands available, we primarily use five commands in Django development:

- **CD** (change down a directory)
- **CD ..** (change up a directory)
- **DIR** (list files in your current directory)
- **MKDIR** (make directory)
- **ECHO>** (create a new file)

Open your command line and try them out. The "**>**" symbol represents our command line prompt for Windows and $ for Mac, and all commands in this guide should be typed after the "**>**" prompt.

For example, assuming you are on a window, let's change into our Desktop directory:

***command prompt***

```
> cd Desktop
```

Now, let's create a new directory folder using 'mkdir,' navigate into it, and add a new file named 'index.html,':

***command prompt***

```
> mkdir django_apps
> cd django_apps
> echo> index.html
```

Next, use "dir" to list all the current files in our directory. You will see that there is just the newly created "index.html":

***command prompt***

```
> dir
12/12/2023  07:46 AM    <DIR>          .
12/12/2023  07:46 AM    <DIR>          ..
12/12/2023  07:46 AM                13 index.html
               1 File(s)             13 bytes
               2 Dir(s)   5,406,769,152 bytes free
```

Advanced developers can efficiently navigate their computers using the keyboard and command line. With practice, using the command line approach is often faster than using a mouse.

While this book provides step-by-step instructions, you do not need to be an expert on the command line. However, developing command-line skills is beneficial for any professional software developer over time.

Two recommended free resources for further study are the Command Line Crash Course and the Codecademy Course on the Command Line.

### Install Python 3 on Mac OS X

While Python 2 comes pre-installed on Mac computers by default, Python 3 is not included. You can verify this by entering the following command in your command line console and pressing Enter:

***mac command prompt***

```
$ python --version
Python 2.7.15
```

To check if Python 3 is already installed, you can run the same command using python3 instead of python:

***mac command prompt***

```
$ python3 --version
```

If your computer displays a version number like 3.7.x (any version of 3.7 or higher), then Python 3 is already installed.
However, if you encounter an error, you'll need to install Python 3 manually. Here are the steps:

First, install Apple's Xcode package:

***mac command prompt***

```
$ xcode-select --install
```

Follow the on-screen instructions to complete the installation. Xcode is a large program, so it may take some time.

Next, install the package manager Homebrew using the following command:

***mac command prompt***

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Confirm that Homebrew is correctly installed by running:

***mac command prompt***

```
$ brew doctor
```

If you receive a message indicating that your system is ready to brew, you're good to go.
Now, let's install the latest version of Python 3:

***mac command prompt***

```
$ brew install python3
```

Confirm the Python 3 installation by running:

***mac command prompt***

```
$ python3 --version
Python 3.11.0
```

To launch a Python 3 interactive shell, which allows you to run Python commands directly, simply type the following in your command line:

***mac command prompt***

```
$ python3
```

You should see a prompt like:

***mac command prompt***

```
Python 3.X.X (default, Month Day Year, HH:MM:SS)
[GCC X.X.X] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

To exit the Python 3 interactive shell at any time, press Control+d (the “Control” and “d” keys simultaneously).
Remember that you can still use Python 2 on your system if needed.

### Install Python 3 on Windows

To check if Python is already installed on your Windows system and, if not, to install it, follow these steps:

**Step 1: Check if Python is already installed**

Press Win + R on your keyboard to open the Run dialog.
Type cmd and press Enter to open the Command Prompt.
In the Command Prompt, type python --version or python -V and press Enter.

If Python is already installed, you'll see the installed version number (e.g., Python 3.8.3).
If Python is not installed, you'll likely see an error message.

**Step 2: Download Python (if not already installed)**

Open your web browser and go to the official Python website at https://www.python.org/downloads/windows/.

On the Python download page, you'll see the latest version of Python for Windows. Click on the "Download Python X.X.X" button, where "X.X.X" is the version number (e.g., Python 3.9.6).

**Step 3: Run the Python Installer**

Once the installer is downloaded, locate the downloaded file (usually in your Downloads folder) and double-click it to run the installer.

In the Python installer window, make sure to check the box that says "Add Python X.X to PATH." This option is essential for easily running Python from the Command Prompt.

Click the "Customize installation" button if you want to customize the installation further. Otherwise, you can proceed with the default settings.

Click the "Install Now" button to start the installation process.

**Step 4: Wait for Installation to Complete**

The installer will now install Python on your system. This may take a few moments.

**Step 5: Verify Python Installation**

After the installation is complete, go back to the Command Prompt by pressing Win + R, typing cmd, and pressing Enter.

Type python --version or python -V again and press Enter.

If Python was installed successfully, you should see the version number you selected during installation.

**Step 6: Test Python**

You can also run a simple Python command to test if it's working. In the Command Prompt, type python and press Enter. You should see a Python shell prompt (usually >>>).

You can type print("Hello, World!") and press Enter. You should see the output "Hello, World!" displayed on the screen.

That's it! You've successfully installed Python on your Windows system and verified that it's working. You can now start using Python for programming and scripting.

### Install Python 3 on Linux

To check if Python is already installed on your Linux system and, if not, to To check if Python is already installed on your Linux system and, if not, to install it, follow these steps::

**Step 1: Check if Python is already installed**

Open a terminal. You can usually do this by pressing Ctrl + Alt + T or by searching for "Terminal" in your desktop environment's applications menu.

In the terminal, type the following command and press Enter:

***terminal***

```
python3 --version
```

If Python is already installed, you'll see the installed version number (e.g., Python 3.8.3).
If Python is not installed, you'll likely see an error message.

**Step 2: Install Python (if not already installed)**

Python is often pre-installed on many Linux distributions. However, if it's not installed, you can install it using your system's package manager.

**Ubuntu/Debian:**

If Python is not already installed, open a terminal and run the following command to install Python 3:

***terminal***

```
sudo apt-get install python3
```

**Fedora:**

If Python is not already installed, open a terminal and run the following command to install Python 3:

***terminal***

```
sudo dnf install python3
```

**CentOS/RHEL:**

Python may not be pre-installed on some CentOS/RHEL systems. To install Python 3, open a terminal and run the following command:

***terminal***

```
sudo yum install python3
```

**Step 3: Verify Python Installation**

After installing Python, you can verify the installation by running the following command in the terminal:

***terminal***

```
python3 --version
```

You should see the version number you installed.

**Step 4: Test Python**

You can also run a simple Python command to test if it's working. In the terminal, type:

***terminal***

```
python3
```

You should see a Python shell prompt (usually >>>).

You can type print("Hello, World!") and press Enter. You should see the output "Hello, World!" displayed on the screen.

That's it! You've successfully installed Python on your Linux system and verified that it's working. If you face any issues, please refer to the following blog:

If you face any issue, please refer to the following blog:
https://solarianprogrammer.com/2017/06/30/building-python-ubuntu-wsl-debian/

You can now start using Python for programming and scripting.

## Virtual environments

Virtual environments are a fundamental and indispensable aspect of modern programming, particularly in the context of Python development.

These environments are essentially isolated containers that house all the necessary software dependencies required for a specific project.

Their significance lies in addressing the challenges posed by managing multiple projects with varying software requirements on a single computer.

The primary reason for using virtual environments, as exemplified in Python programming, is to avoid conflicts between different project dependencies.

In the absence of virtual environments, Python and related packages are typically installed globally on the system.

This poses a significant issue when you need to work on multiple projects simultaneously, each having its own set of software requirements.

For instance, imagine Project A relies on Django version 2.1, while Project B, developed a year earlier, is still dependent on Django 1.10. Without virtual environments, managing such scenarios can become exceedingly complex, but with their implementation, these issues are effortlessly resolved.

One of the key advantages of virtual environments is their ability to keep project dependencies neatly isolated.

Each virtual environment acts as a self-contained ecosystem, allowing you to install and manage specific versions of libraries, frameworks, and packages without affecting the global system environment.

This isolation ensures that changes made to one project's dependencies do not inadvertently impact another project.

It provides a clean slate for each project, eliminating version conflicts and making it easier to maintain and update software components without conflicts.

While, there are several tools available for creating virtual environments in Python, one prominent and officially recommended option is Pipenv.

Introduced by renowned Python developer Kenneth Reitz in 2017, Pipenv has gained widespread popularity due to its simplicity and robustness.

Pipenv operates in a manner akin to npm and yarn in the Node.js ecosystem. It uses a Pipfile to define project dependencies and a Pipfile.lock to guarantee deterministic builds.

Determinism, in this context, means that every time you create a new virtual environment and install dependencies using Pipenv, you will obtain precisely the same configuration.

This predictability is crucial for ensuring that your project behaves consistently across different environments, and when collaborating with other developers.

Deterministic builds minimize the risk of unexpected issues arising from variations in dependency versions, thereby enhancing project stability.

In practice, creating a new virtual environment for each Python project using Pipenv has become the standard practice.

This approach ensures that projects remain self-contained, simplifying development, testing, and deployment processes.

With the help of Pipenv, Python developers can effortlessly manage project-specific dependencies, facilitating smoother project development and collaboration.

To get started with Pipenv, you can use the package manager pip, which is often installed alongside Python 3. The following command demonstrates how to install Pipenv:

***command prompt***

```
> pip install pipenv
```

In conclusion, virtual environments are an essential tool in modern software development, offering isolation and determinism that simplifies project management and ensures reliable and consistent outcomes.

Python's Pipenv, in particular, has become a popular choice for managing virtual environments, making it easier than ever to work on multiple projects with distinct dependencies while maintaining a clean and predictable development environment.

## Install Django

In this section, we will walk you through a series of commands used to set up a new Django project from scratch. These commands will help you create a new project directory, install Django, and start a development server.

**Change Directory to Desktop:**

*command prompt*

```
> cd Desktop
```

This command changes your current working directory to the Desktop. All the subsequent commands will be executed in this location.

**Create a Project Directory:**

The next step is to create a directory where you will store all your Django projects. In this example, we will create a directory called django_apps on your desktop.

You can replace the path with the location where you want to store your projects. Open your command prompt (or terminal on macOS/Linux) and run the following commands:

*command prompt*

```
> mkdir django_projects
> cd django_projects
```

Here, we use the mkdir command to create a new directory called django_apps, and then we use cd to change our current directory to django_ apps.

*command prompt*

```
>mkdir myfirstproject
```

Here, we're creating a new directory called myfirstproject within your Desktop directory. This is where your Django project will reside.

**Change Directory to your Project Directory:**

*command prompt*

```
>cd myfirstproject
```

You navigate into the myfirstproject directory that you just created. This is where you'll be working on your Django project.

**Activate a Virtual Environment:**

*command prompt*

```
>pipenv shell
```

This command activates a virtual environment using Pipenv.

A virtual environment is an isolated environment for your project, ensuring that your project's dependencies don't interfere with other Python projects on your system.

You'll notice that your command prompts changes to include the virtual environment name, in this case, (myfirstproject).

**Install Django:**

***command prompt***

```
> pip install django
```

With the virtual environment activated, you install Django using the pip package manager. This step ensures that your project has Django available as a dependency.

**Start a New Django Project:**

***command prompt***

```
> django-admin startproject myfirstproject .
```

Here, you use the django-admin command to create a new Django project named myfirstproject. The . at the end specifies that the project should be created in the current directory.

This command sets up the initial structure for your Django project.

**Run the Development Server:**

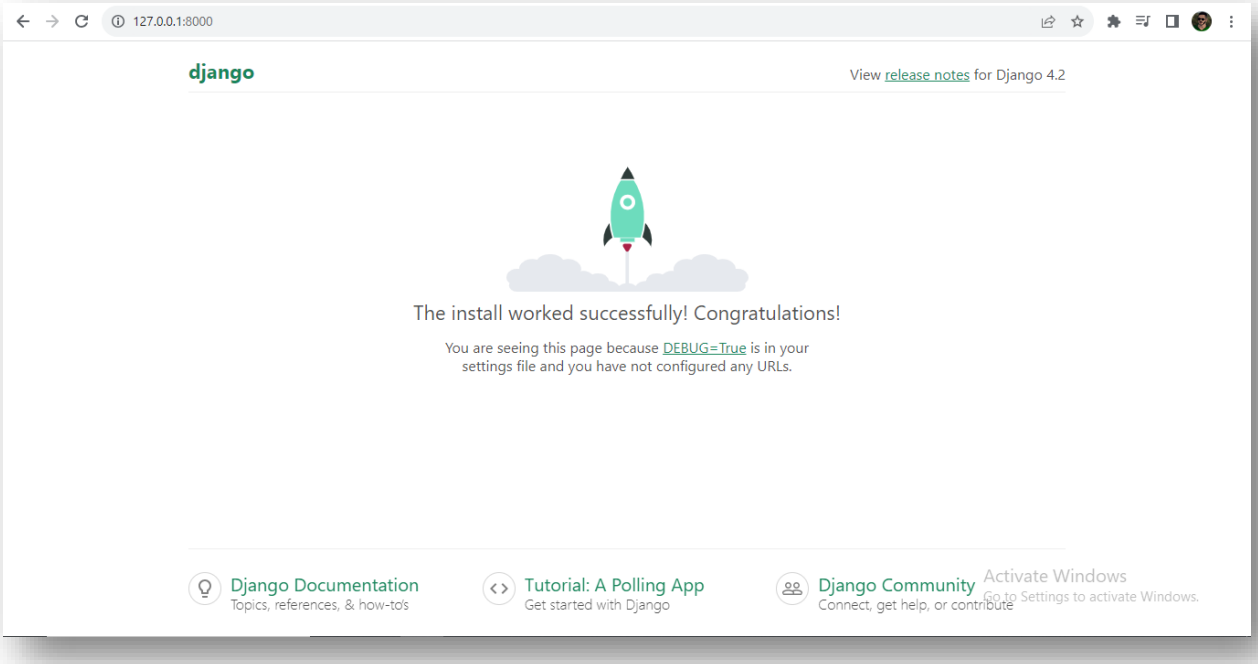
***command prompt***

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

Finally, you use the manage.py script provided by Django to start the development server. This server allows you to preview your Django project locally.

The server will be accessible http://127.0.0.1:8000/.

You can open a web browser and visit the provided URL to see the default Django welcome page, indicating that your project is up and running.




These commands are the initial steps to create a Django project and start developing your web application.

As you continue your journey with Django, you'll use additional commands to create apps, define models, and build the functionality of your web application.

## Understanding the Django Project Structure

In this section, we'll dive deep into the Django project structure, unveiling the various files and directories that make up a Django project.

Understanding the project structure is crucial for any Django developer, as it forms the foundation of your web application.

Let's embark on this journey to explore the inner workings of a Django project.

### Overview

A Django project is organized into a well-defined directory structure that follows the "Django way" of development.

This structure encourages modularity, clean code, and separation of concerns.

Before we delve into the details, let's take a look at the high-level overview of a typical Django project directory:

```
myproject/
   ├── myproject/
   │   ├── __init__.py
   │   ├── settings.py
   │   ├── urls.py
   │   └── wsgi.py
   ├── manage.py
   ├── app1/
   ├── app2/
   └── ...
```

Let's break down each of these components step by step.

**The Project Directory**

**myproject/**
The outermost directory, often named after your project, contains all the project-related files and directories.

The project name is user-defined and may vary from one project to another. It's good practice to choose a name that reflects the purpose of your project.

**myproject/__init__.py**
This empty file is required to tell Python that this directory should be treated as a Python package. It's often empty but can contain package-level initialization code if necessary.

**myproject/settings.py**
The heart of your Django project, this file holds all the project settings. You'll configure your database, installed apps, middleware, authentication settings, and much more in this file.

**myproject/urls.py**
This file defines the URL patterns for your project and serves as a roadmap for how incoming requests are routed to the appropriate view functions within your apps.

**myproject/wsgi.py**
This file is used for deploying your Django application on a production server. It serves as an entry point for the WSGI (Web Server Gateway Interface) server to communicate with your Django app.

**The manage.py Script**

The manage.py script is a powerful tool that simplifies various tasks related to your Django project.

You can use it to create database tables, run development servers, create superusers, and much more. The manage.py script is a Python script that interacts with your project's settings.

**The Apps**

Django encourages a modular approach to development. Each logical component of your project is encapsulated within a Django app.

These apps can be reused across projects, fostering code reusability.

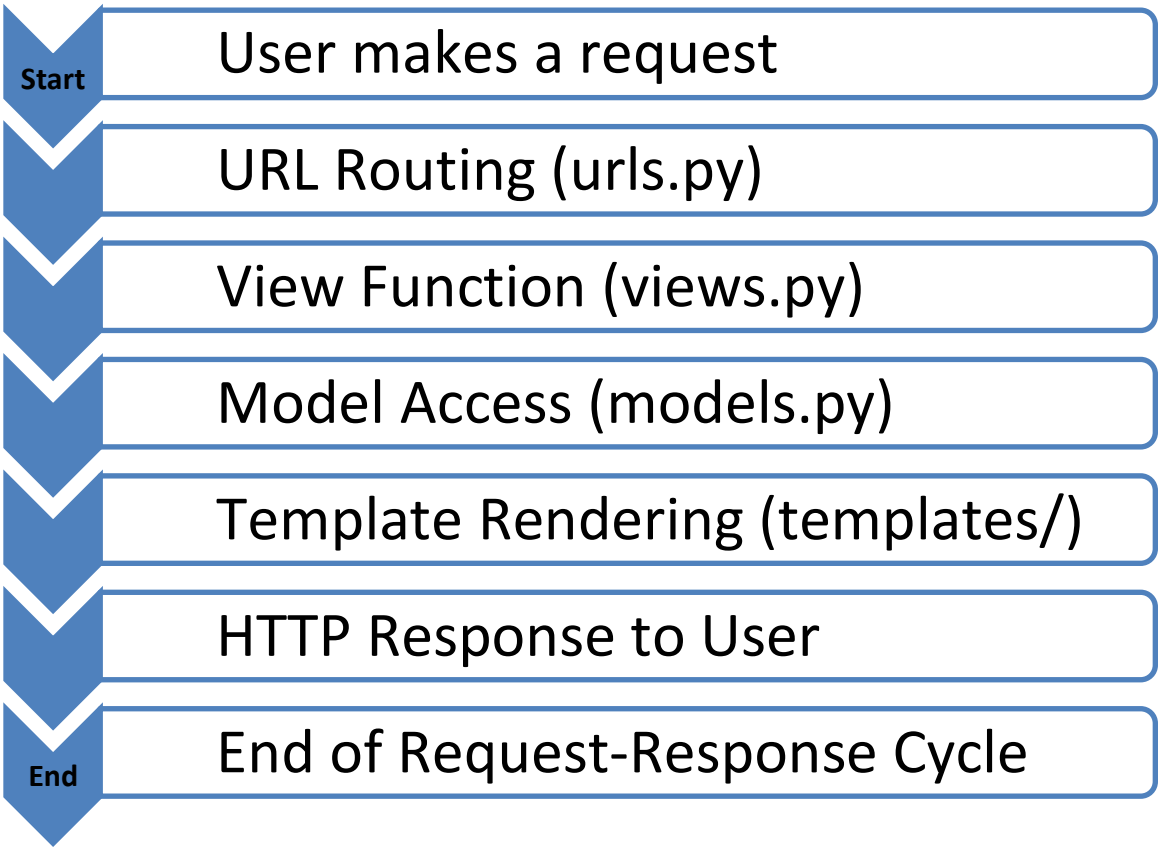
Examples of apps include user authentication, blog functionality, or an e-commerce system. You can create your apps using the Python manage.py startapp app_name command.

Understanding the Django project structure is the first step in becoming a proficient Django developer.

In the next section, we'll delve into configuring your project settings and kickstart your Django journey!

**FlowChart**

Designing a comprehensive flowchart for the entire Django framework can pose a challenge due to its modular and adaptable characteristics. Presented here is a simplified flowchart that delineates the fundamental request-response cycle within a Django application. It is essential to note that this representation provides a high-level overview, while the actual flow includes additional intricacies and interactions.




**User Initiates Request:** The process commences when a user triggers an HTTP request within the Django application, typically through actions like clicking a link, submitting a form, or engaging in other interactive elements.

**URL Routing (urls.py):** Django utilizes the defined URL patterns in the urls.py file to ascertain which view function should handle the incoming request.

**View Processing (views.py):** The view function takes charge of processing the request, engaging with models if necessary, and determining the data to be presented. Subsequently, it produces an HTTP response.

**Model Interaction (models.py):** If database interaction is required, the view communicates with model classes specified in models.py. This involves tasks such as creating, retrieving, updating, or deleting data in the database.

**Template Rendering (templates/):** To generate dynamic content, the view employs template rendering. Templates, comprised of HTML files with embedded Django template language syntax, enable the incorporation of dynamic data.

**User Receives HTTP Response:** The rendered template becomes part of an HTTP response, which is dispatched back to the user's browser. The response may contain HTML, JSON, or other content types based on the nature of the request.

**Conclusion of Request-Response Cycle**: The cycle concludes, and the user observes the outcome of their request in their browser.

It's important to acknowledge that this depiction is a simplified overview, and real-world scenarios may involve additional components, middleware, and configurations in the overall process.

Here is an example:

Django, a Python web framework known for fostering rapid development and adhering to a clean, pragmatic design philosophy, follows the Model-View-Template (MVT) architectural pattern, a variation of the traditional Model-View-Controller (MVC) pattern. Here's a simplified breakdown of Django's functioning:

**Models:**

Within Django, a model takes the form of a Python class representing a database table, with each class attribute corresponding to a field in the respective table.

Models serve to define the structure of data and provide an abstraction layer for interacting with the database.

Django employs an Object-Relational Mapping (ORM) system, enabling interaction with the database through Python code rather than direct SQL queries.

**Views:**

Views in Django play a crucial role in handling user requests and delivering corresponding responses. They take input from users in the form of HTTP requests, interact with required data (often through models), and subsequently generate responses, which may manifest as HTML pages, JSON, or other formats. Views encapsulate the application's business logic and manage the flow of data within the framework.

**Templates:**

Templates in Django manage the presentation logic by dynamically generating HTML based on data supplied by views. Utilizing a distinct syntax, templates enable the embedding of Python-like code directly within HTML for seamless integration of dynamic content.

**URLs and Routing:**

Within Django, a URL routing mechanism is employed to link URLs to specific views.

In the urls.py file, you establish patterns that align with particular URLs and link them to their corresponding views.

**Settings and Configuration:**

Django employs a settings module to store configuration parameters for your project, encompassing aspects like database settings, middleware, and more.

These configurations are housed in the settings.py file within your project.

**Middleware:**

Middleware components serve to globally process requests and responses, either before they reach the view or after they leave it.

Instances of middleware include authentication middleware, security middleware, and others.

Django furnishes a toolkit and established conventions that simplify the construction of web applications, fostering code reusability, modularity, and maintainability. Adhering to these conventions and patterns allows developers to concentrate on feature development, rather than grappling with low-level intricacies.

## Install Git

In the world of web development, managing your project's codebase is essential.

You need a system that allows you to keep track of changes, collaborate with other developers seamlessly, and ensure that your Django web applications are always in a reliable state.

This is where Git comes into play.

### What is Git?

Git is a distributed version control system that helps you track changes in your codebase. It was created by Linus Torvalds in 2005 and has since become an indispensable tool for software development. Git is particularly valuable in the Django ecosystem for several reasons:

- **Version Control:** Git allows you to track every change made to your project, making it easy to roll back to previous states or identify when and why a particular change was made.

- **Collaboration:** When working on a Django project with a team, Git enables smooth collaboration. Multiple developers can work on the same codebase simultaneously without interfering with each other's work.

- **Branching and Merging:** Git's branching and merging capabilities are powerful tools for managing different features or bug fixes in isolation. You can develop new features in separate branches and then merge them into the main codebase when they are ready.

- **Backup and Recovery**: Git provides a secure and efficient way to back up your project. If something goes wrong, you can always revert to a known working state.

- **Open Source:** Git is open source and widely adopted, which means a vast community of developers continually improves and supports it.

### Installing Git

Before you can start using Git with your Django project, you'll need to install it on your development machine. Here's how to do it:

**On Windows:**

Visit the official Git website at https://git-scm.com/.

Download the Windows installer and run it.

Follow the installation wizard's instructions, accepting the default settings unless you have a specific reason to change them.

After installation, open the Git Bash terminal that comes with Git, or use your preferred terminal if you're more comfortable with that.

**On macOS:**
If you have Homebrew installed, you can install Git by running the following command in your terminal:

***mac command prompt***

```
brew install git
```

If you don't use Homebrew, you can download the macOS Git installer from https://git-scm.com/download/mac and follow the installation instructions.

Once installed, open your terminal, and Git should be available for use.

**On Linux (Ubuntu/Debian):**
Open a terminal.

Run the following command to install Git:

***Terminal***

```
sudo apt-get update
sudo apt-get install git
```

After installation, Git will be ready to use in your terminal.

**Verifying the Installation**
To verify that Git has been installed successfully, open your terminal and run the following command:

***Terminal***

```
$ git --version
```

If Git is installed correctly, you should see the installed version displayed in the terminal.

Congratulations! You now have Git installed on your system and are ready to start using it to manage your Django projects more effectively. In the following chapters, we'll explore how to initialize a Git repository, make commits, create branches, and collaborate with others using Git.

## Text Editors

**Text Editors and Setting Up Visual Studio Code for Django.**
A text editor is a software application that allows you to create, edit, and manage the code that makes up your Django projects. Choosing the right text editor is crucial as it significantly impacts productivity and coding experience. In this chapter, we'll explore what text editors are, recommend a few popular ones, and guide you through the process of downloading and installing Visual Studio Code (VS Code), a widely used and versatile text editor for Django development.

**What Are Text Editors?**

Text editors are software tools designed specifically for working with plain text files, including code files written in programming languages like Python, HTML, CSS, and JavaScript. They provide an environment for developers to write, edit, and organize code efficiently. Unlike full-fledged integrated development environments (IDEs), text editors are lightweight and flexible, making them an excellent choice for web development with Django.

**Why Do You Need a Text Editor?**

Here are some reasons why you need a text editor for Django development:

- **Code Writing:** Text editors are optimized for writing code, offering features like syntax highlighting, auto-indentation, and code completion that make coding faster and more accurate.

- **Customization:** Text editors are highly customizable, allowing you to install extensions and configure settings to suit your development workflow and coding style.

- **Lightweight:** Text editors are generally less resource-intensive than IDEs, making them more responsive, especially on older computers.

- **Versatility:** Text editors are not limited to Django development; you can use them for various programming tasks and languages.

Now, let's dive into one of the most popular text editors for Django development: Visual Studio Code (VS Code).

## Visual Studio Code (VS Code)

Visual Studio Code, often referred to as VS Code, is a free and open-source text editor developed by Microsoft. It has gained immense popularity among developers due to its extensibility, powerful features, and a large community of users and extensions.

**Downloading and Installing VS Code**

Here's how you can download and install VS Code:

Visit the Official Website: Open your web browser and go to the official Visual Studio Code website at https://code.visualstudio.com/.

Download for Your Platform: VS Code is available for Windows, macOS, and Linux. Click on the download link corresponding to your operating system.

**Install VS Code:**

**Windows:** Once the installer is downloaded, run it and follow the installation wizard's instructions.

**macOS:** Locate the downloaded file in your Downloads folder, double-click it, and follow the installation instructions.

**Linux:** Depending on your distribution, you may need to follow specific installation instructions.

*Refer to the VS Code documentation for details.*

**Launch VS Code:** After installation, you can launch VS Code from your system's application launcher or by running the code command in your terminal (Linux).

**Basic Configuration:** Upon first launch, you can customize VS Code to your liking by installing extensions and adjusting settings. VS Code will guide you through this process.

### Recommended VS Code Extensions for Django Development

To enhance your Django development experience with VS Code, consider installing the following extensions:

- **Python:** Provides Python language support, including syntax highlighting, debugging, and code completion.
- **Django:** Offers Django-specific features like template language support and code snippets.
- **Pylance:** Enhances Python support in VS Code, providing advanced features like type checking and code analysis.
- **GitLens:** Improves Git integration within VS Code, helping you manage version control effectively.

With your VS Code installation and extensions set up, you're well-prepared to embark on your Django development journey.
In the upcoming chapters, we'll delve deeper into using VS Code for Django projects and explore various aspects of web development with this powerful framework.

Occasionally, you may observe incorrect Python code indentation due to width constraints on this page. To rectify this issue, kindly copy the code and paste it into a text editor such as VS Code. Doing so should result in the correct indentation.

# Chapter 3: Views and Templates

## Creating a New Django Project

In this chapter, we will go through the process of setting up our development environment and creating our first Django project.
By the end of this chapter, we will have a basic understanding of how to create a Django project and run it locally.

**Step 1: Creating a Project Directory**
The first step is to establish a directory where we can store all our Django projects. In this example, we will create a directory named "django_projects" on our desktop.

You can replace the path with your preferred location for storing your projects.
Now, open the command prompt (or terminal on macOS/Linux) and execute the following commands:

***command prompt***

```
> cd Desktop
> mkdir django_projects
> cd django_projects
```

Here, we use the mkdir command to create a new directory called "django_projects," and then we use cd to change our current directory to "django_projects."

**Step 2: Creating a Django Application Directory**
Within the "django_projects" directory, we need to establish a directory for our specific Django project. Let's name it "newspaper_app."

Execute the following commands:

***command prompt***

```
> mkdir newspaper_app
> cd newspaper_app
```

Once again, we use the mkdir command to create a new directory called "newspaper_app," and then we use cd to change our current directory to "newspaper_app."

**Step 3: Setting Up a Virtual Environment**
Now, it's time to set up a virtual environment for our Django project.
A virtual environment is an isolated environment that allows us to install packages and dependencies separately for each project, ensuring that project-specific dependencies don't interfere with each other. We will use pipenv, a popular tool for managing virtual environments and dependencies.

Run the following command:

***command prompt***

```
> pipenv shell
```

This command creates and activates a virtual environment named 'newspaper' for our project. You can replace 'newspaper' with your preferred name. Once activated, any packages installed will be isolated within this environment.

**Step 4: Installing Django**

Now that we have our virtual environment set up, we can proceed to install Django.
Run the following command:

***command prompt***

```
> pip install django
```

This command installs Django within our virtual environment, making it available for our project.

**Step 5: Creating a Django Project**

With Django installed, we can now create a new Django project.
Run the following command:

***command prompt***

```
> django-admin startproject newspaper_project .
```

Here, we use django-admin to initiate a new Django project named "newspaper_project." The "." at the end specifies that the project should be created in the current directory. This command generates the basic structure and files needed for a Django project.

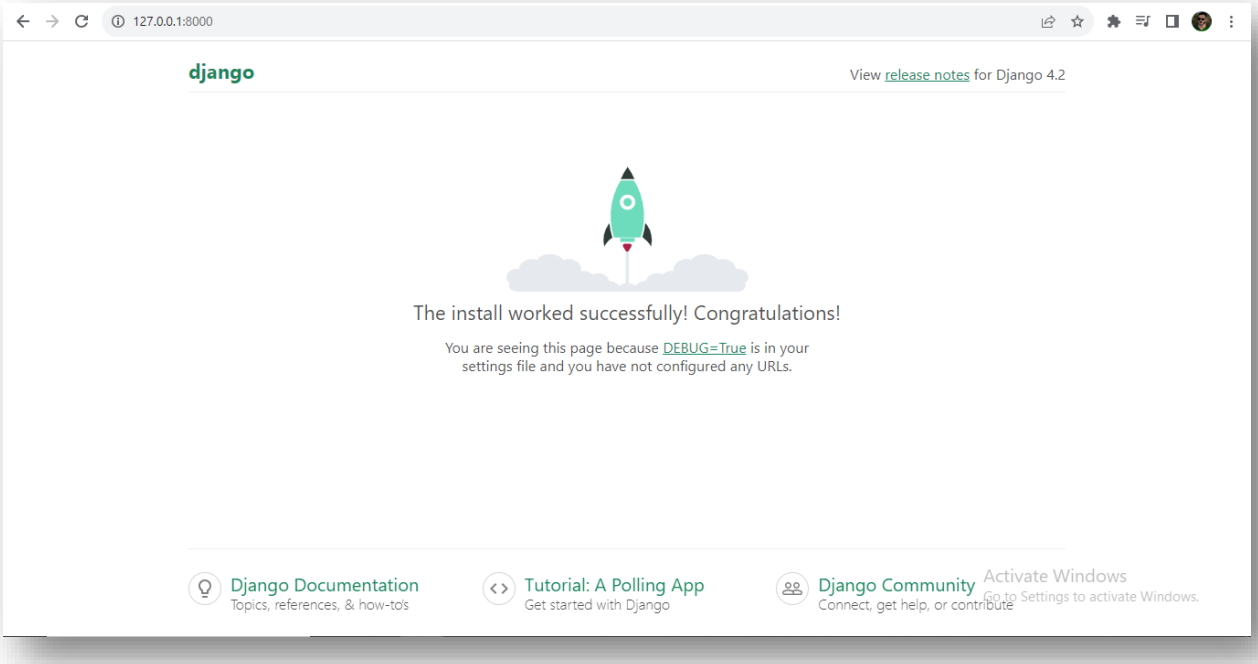
**Step 6: Running the Development Server**

Finally, let's start the development server to see our Django project in action.
If the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

This command launches the Django development server, and we should see output indicating that the server is running.
We can now open a web browser and visit http://127.0.0.1:8000/ to view our new Django project's welcome page.




Congratulations! we've successfully set up our Django development environment, created a new project, and started the development server.

## Creating a New App - The News App

Creating an app in a Django project is a fundamental step in structuring and organizing our web application.

Django, a high-level Python web framework, is renowned for its adherence to the "Don't Repeat Yourself" (DRY) principle and employs the "Model-View-Controller" (MVC) architectural pattern, known as "Model-View-Template" (MVT) in Django.

## Benefits of Creating Apps:

- **Modularity and Reusability:** Django apps are modular components that encapsulate specific functionality within our web application. This modular approach simplifies code organization and encourages reusability across different projects.
- **Separation of Concerns:** Apps enable the separation of different aspects of our application, such as user authentication, blog posts, or e-commerce features. This separation enhances code clarity and organization.

- **Code Organization:** Django enforces a structured directory layout within each app, aiding developers in locating and working with different parts of the application. This structure includes directories for models, views, templates, static files, and more.
- **Collaboration:** When collaborating in a team, dividing our project into apps promotes teamwork by allowing team members to focus on different app components, minimizing conflicts in a shared codebase.
- **Scalability:** Django apps are designed for scalability. As our project grows, we can effortlessly introduce new apps to manage additional features without disrupting the existing codebase.
- **Reusable Apps:** Django's ecosystem offers a wide array of third-party apps and packages that can be seamlessly integrated to add common functionalities, saving valuable development time.
- **Testing and Maintenance:** Apps can be individually tested, simplifying issue identification and resolution. This modular approach also streamlines maintenance efforts, as specific apps can be updated or replaced without affecting the entire project.
- **Encapsulation and Isolation:** Apps provide a level of encapsulation and isolation, allowing for the independent management of dependencies, templates, and static files for each app.

**Exiting the Running Server:**
To exit the currently running server, follow these steps:
Open the command window.
Press **Control + C**.

### Creating a new Django app named 'news':

To create a new Django app named 'news,' execute the following command:

***command prompt***

```
> python manage.py startapp news.
```

In Django, an app is a modular component that encapsulates a specific functionality within the application. In this case, we are creating the 'news' app to manage news-related content in our newspaper application.

### Configuring installed apps

To configure our Django project's installed apps, we need to work with the INSTALLED_APPS list located in the settings.py file. This list determines which applications are active within our project.

Occasionally, you may observe incorrect Python code indentation due to width constraints on this page. To rectify this issue, kindly copy the code and paste it into a text editor such as VS Code. Doing so should result in the correct indentation.

***settings.py***

```
INSTALLED_APPS = [
```

```
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'news',  # Add the 'news' app here.
]
```

This list includes core Django functionality with prefixes like django.contrib, offering features such as an admin interface, authentication, content types, sessions, and message handling. We can include or exclude these apps based on our project's needs.

Additionally, we can include custom apps like 'news' in the INSTALLED_APPS list. This indicates that we've either created a 'news' app ourselves or included one created by someone else, and we want to integrate it into our project.

Adding an app to INSTALLED_APPS, allows Django to recognize and use it within our project. This step is crucial for organizing our project's functionality into manageable and scalable components, known as apps.

The 'news' app likely contains models, views, and templates designed for handling news articles or related content. Including it in INSTALLED_APPS ensures Django can recognize and utilize these components.

Before proceeding, let's open Visual Studio Code (VS Code) and select the appropriate interpreter. It is crucial to choose the correct interpreter as, since we have already downloaded Django and other dependencies in the virtual environment. Therefore, we need to instruct VS Code on where to locate and interpret the code.

So we have to point the interpreter to our virtual environment.

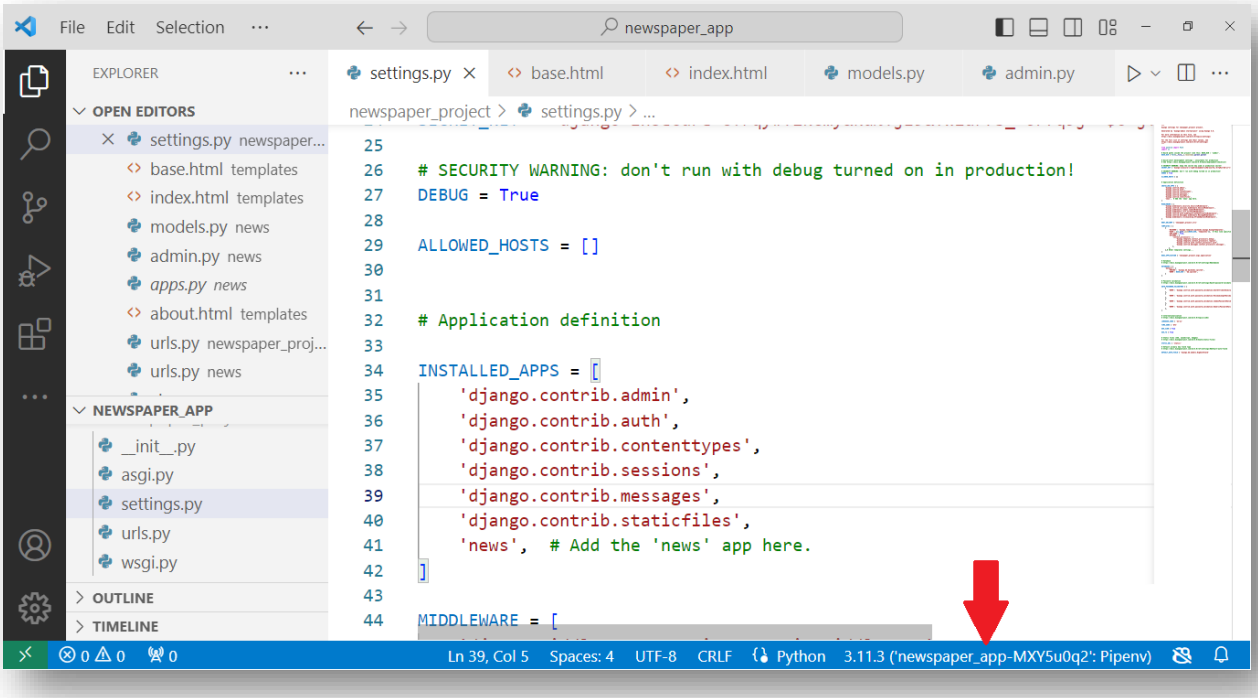
Select Python Interpreter:

- Open the Command Palette by pressing Ctrl+Shift+P.
- Type and select "Python: Select Interpreter."
- A list of detected Python interpreters will be displayed. Choose the one we want to use for our project.

Manually Specify Interpreter Path (Optional):

- If the interpreter we want to use is not automatically detected, we can manually specify the interpreter path.
- Open the Command Palette (Ctrl+Shift+P) and type "Python: Select Interpreter."

- Choose "Enter interpreter path", and provide the path to our Python interpreter executable.




After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

This step reloads the server with the news app and URL configurations, enabling us to define views and templates for the 'news' app, and access it through the development server.

## Views

### Creating Our First View

To create our first view, follow these steps:

Go to the 'news/views.py' file.

Paste the following code:

**news/*views.py***

```
# We're creating a view function, which is like a handler for web requests.
# First, we import some tools from Django that we need for our view.

from django.http import HttpResponse  # A class for creating HTTP responses

# Now, let's define our view function. It's called newsView, and it takes a
'request' as its parameter.
```

```
def newsView(request):

    # Inside the function, we're keeping things simple. We're just returning an
HTTP response.

    # In this case, it's a response saying 'Hello, World!'

    return HttpResponse('Hello, World!')
```

The HttpResponse class is what we're using to send a response back to whoever made a request to our web application.Our newsView function is quite straightforward. When someone accesses a specific URL in our web app, this function gets called.

It doesn't do anything fancy; it just responds with 'Hello, World!'. Views are fundamental to web applications, as they handle user requests, execute logic, interact with the database, and generate responses. In more complex applications, this function might interact with a database, render dynamic web pages, or do a lot of other cool things.

But for now, it's keeping it simple and just saying hi!

## Urls

**Setting Up URL Routing in Django for our Newspaper App**

In this section, we'll delve into Django's URL routing mechanism, which enables us to connect URLs to view functions.

Creating an 'urls.py' File for the 'news' App:

To create an empty 'urls.py' file within the 'news' app directory, use the following command:

***command prompt***

```
>echo >news/urls.py
```

The 'urls.py' file for the 'news' app defines URL patterns and routing. URL mappings link views to specific URLs, which dictate the displayed content.

In Django, URL routing directs incoming requests to view functions, enabling the creation of dynamic and interactive web applications.

***news/urls.py***

```
# We start by importing the necessary module for handling URL patterns in Django.
```

```python
from django.urls import path

# Next, we import the 'newsView' function from the 'views' module of the current
Django app (denoted by the dot).

from .views import newsView

# Now, we define a variable called 'urlpatterns'. This variable will hold a list
of URL patterns for our Django app.

urlpatterns = [

    # Inside the list, we use the 'path' function to define a URL pattern.
    # The first argument ('') represents the URL path. An empty string ('') means
this pattern matches the root URL.
    # The second argument ('newsView') is the view function that will be called
when this URL pattern is matched.
    # The third argument ('home') is a unique name given to this URL pattern,
which can be used to reference it later in the code.

    path('', newsView, name='home')

]
```

In simpler terms:

**Importing Modules:**
We're importing the necessary tools from Django to handle URL routing (path) and importing our newsView function.

**URL Patterns List:**
We create a list called urlpatterns to store our URL patterns.

**Defining a URL Pattern:**
We use the path function to define a URL pattern.

- The first argument '' means this pattern matches the root URL (e.g., http://example.com/).
- The second argument newsView is the function that will be executed when this URL is visited.
- The third argument name='home' is a name we give to this pattern, which helps us reference it later in the code.

This code essentially says, "Hey Django, when someone visits the root URL of our app, call the newsView function, and we'll call this pattern 'home'."

**urls.py in the newspaper_app**

**newspaper_*projects/urls.py***

```
# This line imports the admin module from the django.contrib package.
from django.contrib import admin

# This line imports the path and include functions from the django.urls package.
from django.urls import path, include

# This is a list called urlpatterns, which is used to define the URL patterns for
our Django project.
urlpatterns = [
    # This line maps the URL 'admin/' to the Django admin site. When we go to
'/admin/', we access the admin interface.
    path('admin/', admin.site.urls),

    # This line maps the root URL ('') to the 'news.urls' module.
    # It means that any request to the root URL of our project will be handled by
the URLs defined in the 'news.urls' module.
    # This is a way to organize and modularize our URL patterns.
    path('', include('news.urls')),
]
```

So, in summary, this code sets up the URL patterns for a Django project. It includes the default admin interface at '/admin/' and delegates any other URL patterns to be handled by the URLs defined in the 'news.urls' module. This modular approach makes it easier to manage and organize our project's URL structure.

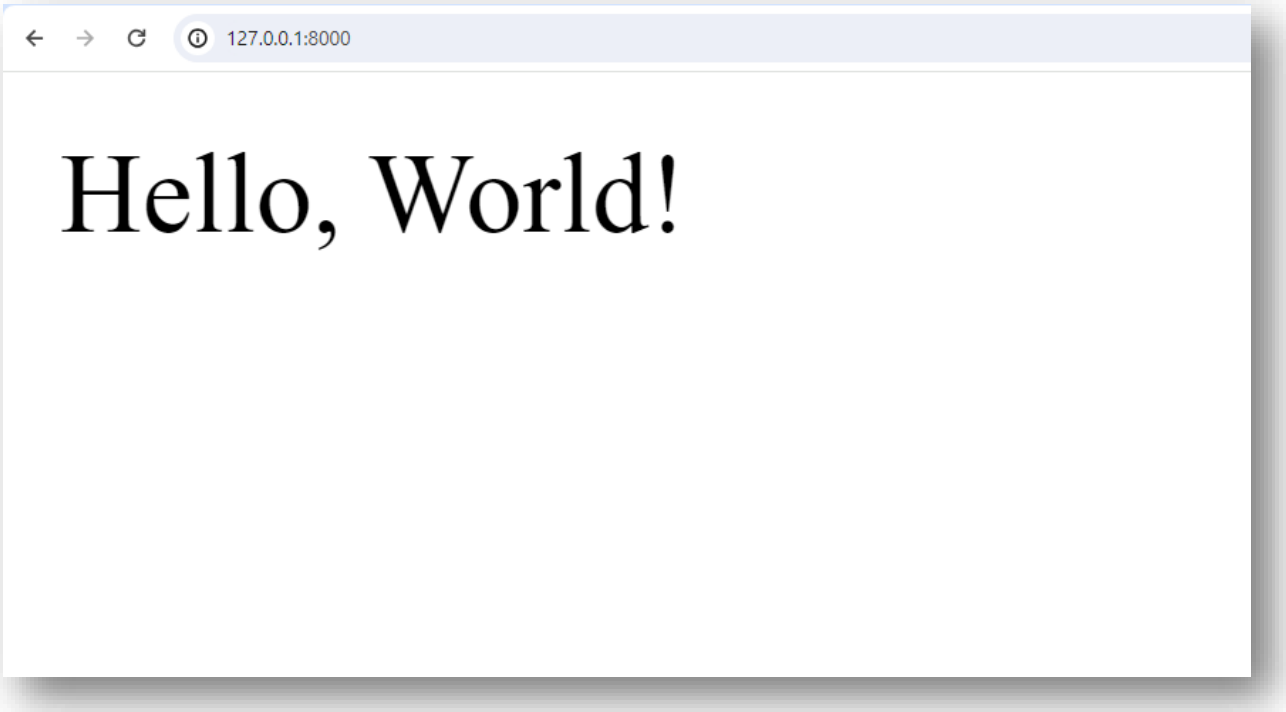
URL routing in Django is essential for creating well-organized, maintainable, and scalable web applications. It segregates the logic of handling different URLs into distinct views, enhancing code modularity. By associating URLs with views, we can ensure that each page of our website is managed by the appropriate function, facilitating the creation of complex web applications.

Furthermore, it permits the reuse of apps across different projects, as their URL patterns can be included wherever needed.After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

This command starts the Django development server.



## Templates

**Creating a 'template' Directory:**

To begin, let's create a 'template' directory in the project's root directory. This directory will serve as the storage space for HTML templates, which are essential for rendering web pages in our Django application.

To create this directory, use the following command:

***command prompt***

```
>mkdir template
```

**Creating an 'index.html' File:**

Now, let's create an empty 'index.html' file within the 'template' directory.HTML templates define the structure and layout of web pages in a Django application. This file will be populated with HTML code later.

Execute the following command to create the 'index.html' file:

***command prompt***

```
>echo> template/index.html
```

**Configuring Templates:**

Moving forward, let's delve into the TEMPLATES setting in Django.This setting plays a crucial role in determining how Django handles the rendering of HTML templates for our views. It's essential for building the user interface of our web application.

Now, let's define the path to our templates file in settings.py.

***settings.py***

```
from pathlib import Path
import os
```

**import os**: Import the Python os module at the beginning of your settings.py file. The os module is essential for handling operating system-related tasks. Placing the import statement at the top of the file ensures that all subsequent code can make use of the os module.

Now, let's update the TEMPLATES section in settings.py.
Here's the relevant portion of the code:

***settings.py***

```
TEMPLATES = [
    {

        # Other templates settings...
```

```
        'DIRS': [os.path.join(BASE_DIR, 'templates')],   # This line specifies
the template directory path.

        # Other templates settings...
  },
]
```

In this code snippet, we are primarily focusing on the 'DIRS' key within the template settings dictionary. This key points to a list of directories where Django should search for template files.Storing templates in a dedicated 'templates' directory helps maintain project organization and aligns with best practices for Django development.

Properly configuring INSTALLED_APPS and templates is a foundational step in setting up a Django project, enabling the development of robust and maintainable web applications.

**Creating a Basic View:**
In Django, views are responsible for handling incoming requests, processing data if necessary, and returning an appropriate response.

Let's begin by creating a simple view that displays the homepage of our website.To create views in Django, define Python classes. In our views.py file, we should have a structure similar to the following:

***news/views.py***

```
# We're importing a class named TemplateView from the django.views.generic
module.
from django.views.generic import TemplateView

# We're defining a new class called HomePageView, and it's inheriting from
TemplateView.
class HomePageView(TemplateView):
    # Here, we're setting a property of the class called template_name.
    # This property tells Django which HTML template to use when rendering this
view.
    # In this case, it's set to 'index.html', so Django will look for a file
named 'index.html'.
    template_name = 'index.html'
```

**In simpler terms:**

- We import a special class called TemplateView from Django that helps in rendering HTML templates.

- We create a new class called HomePageView, and we say that it behaves like a TemplateView.
- Inside our new class, we specify the HTML template that should be used when someone visits the home page. The template is named 'index.html'.

This code essentially sets up a basic view for the homepage of a website using Django, and it says, "When someone goes to the homepage, show them the content from the 'index.html' file."

## Adding an About Page to our Django Project

**Step 1: Create the About Page Template**

First, we need to create the HTML template for our About Page. Open our command prompt or terminal and navigate to our project directory. Use the following command to create the template:

***command prompt***

```
> echo > templates/about.html
```

This command will create an empty HTML file named "about.html" in the "templates" directory of our project.

**Step 2: Define the About Page View**

In our project's views.py file, we will define a view class for the About Page. This view will render the "about.html" template. Here's how we can define the view:

**news/views.py**

```
from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = 'index.html'


class AboutPageView(TemplateView): #new
    template_name = 'about.html'
```

Make sure to import the necessary modules at the top of our views.py file.

**Step 3: Update the URLs**

To make our About Page accessible via a URL, we need to update our project's urls.py file. If our project has a separate app for news (as indicated by "Urls.py (news app)"), add the following import statement to our urls.py file:

**from .views import AboutPageView**

Next, update the urlpatterns list to include a URL pattern for the About Page. Replace any existing patterns if needed. Here's an example:

**news/urls.py**

```
from django.urls import path
from .views import HomePageView, AboutPageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
    path('about/', AboutPageView.as_view(), name='about'),
]
```

This code snippet adds a URL pattern that associates the "/about/" URL with the AboutPageView.

**Step 4: Start the Web Server**

After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

Once the server is running, open our web browser and navigate to **http://127.0.0.1:8000/about**. We will now be able to see our newly created "About Page."

That's it! We have successfully added an "About Page" to our Django project. This page can now be used to provide information about our project or application to our users.

**Extending Templates**

In our Django project, follow these steps to create a base HTML template and include it in our home and about pages. Create a new file called base.html in the templates directory:

***command prompt***

```
> echo > templates/base.html
```

After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

In our HTML templates, we can use the {% extends %} and {% block %} tags to inherit from the base template and define content specific to each page. Here's how our HTML templates should look:

**templates/base.html**

```
<!-- This is the header section, common to all pages -->
<header>
  <!-- These links use the Django template tag {% url %} to dynamically generate URLs -->
  <a href="{% url 'home' %}">Home</a> | <a href="{% url 'about' %}">About</a>
</header>

<!-- The following lines use the Django template tags {% block %} and {% endblock %} -->
<!-- They define a block named 'content', which can be overridden in child templates -->
{% block content %}
{% endblock content %}
```

**templates/index.html**

```
<!-- This line indicates that this template extends the 'base.html' template -->
{% extends 'base.html' %}

<!-- Here, we override the 'content' block defined in 'base.html' -->
{% block content %}
  <!-- The content specific to the 'home' page goes here -->
  <h1>Homepage</h1>
{% endblock content %}
```

**templates/about.html**

```
<!-- This line indicates that this template extends the 'base.html' template -->
{% extends 'base.html' %}

<!-- Here, we override the 'content' block defined in 'base.html' -->
{% block content %}
  <!-- The content specific to the 'about' page goes here -->
  <h1>About page</h1>
{% endblock content %}
```

Now, let me explain in simpler terms:

**base.html:**

- The base template contains a header with links to the 'Home' and 'About' pages.
- It also defines a block named 'content', a kind of placeholder for the specific content of each page.

**index.html and about.html:**

- Both of these templates extend the 'base.html' template. This means they include everything from 'base.html'.
- They override the 'content' block with their specific content.
- In 'index.html', there's an <h1> tag with "Homepage" as content.
- In 'about.html', there's an <h1> tag with "About page" as content.

So, essentially, this structure allows us to create a consistent layout across multiple pages (thanks to 'base.html') while customizing the unique content of each page using the 'content' block. It's a handy way to manage a website with a uniform structure.

After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

We can access our webpages at http://127.0.0.1:8000/ and http://127.0.0.1:8000/about. We'll notice that the header is automatically included in both locations.

**Home Page:**


**About Page:**


These changes enable us to maintain a consistent header across multiple pages in our Django project, making our website more organized and user-friendly.

## Tests

Let's delve into the topic of tests. Even in the context of this fundamental application, it remains crucial to incorporate testing and cultivate the habit of consistently applying it to our Django projects. In the words of Jacob Kaplan-Moss, one of Django's original creators, "Code without tests is broken as designed."

The significance of writing tests lies in the automation of the process to validate that the code functions as anticipated. While in a simple application like this one, we could manually inspect the home and about pages to ensure they exist and contain the intended content. This approach becomes impractical as Django projects expand in size. With potentially hundreds or even thousands of individual web pages, manually inspecting each page becomes unfeasible. Moreover, when making alterations to the code—whether it involves introducing new features, updating existing ones, or removing unused sections of the site—it is crucial to confirm that such changes do not inadvertently disrupt other parts of the site. Automated tests empower us to articulate our expectations for a specific aspect of our project once, and subsequently, delegate the verification process to the computer.

Thankfully, Django is equipped with robust, built-in testing tools for crafting and executing tests.

if you explore the news app, you'll discover that Django has already provided a tests.py file that we can leverage. Open it and incorporate the following code:

**tests.py**

```
from django.test import TestCase
from django.urls import reverse

class HomePageViewTest(TestCase):
    def test_home_page_status_code(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_home_page_template_used(self):
        response = self.client.get(reverse('home'))
        self.assertTemplateUsed(response, 'index.html')

class AboutPageViewTest(TestCase):
    def test_about_page_status_code(self):
        response = self.client.get(reverse('about'))
        self.assertEqual(response.status_code, 200)

    def test_about_page_template_used(self):
        response = self.client.get(reverse('about'))
        self.assertTemplateUsed(response, 'about.html')
```

Within this context, we opt for TestCase, but we can also use SimpleTestCase since we're not utilizing a database. In cases where a database is employed, we would have to employ TestCase. The code then proceeds to verify if the status code for each page is 200, the standard response for a successful HTTP request. This essentially ensures the existence of a given webpage but does not make any assertions about the content on that page—a sophisticated way of affirming the page's existence without specifying its content.

The reverse function is used to generate the URLs for the views based on their names specified in the urlpatterns. The client attribute is an instance of Django's test client, which is used to simulate HTTP requests.

To execute the tests, stop the server using Control+c, and enter the command "python manage.py test" in the command line:

***command prompt***

```
> python manage.py test
```

You will get a similar output as the following:
Output:

***command prompt***

```
Found 4 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------------------------
Ran 4 tests in 0.044s

OK
Destroying test database for alias 'default'...
```

Great job! In the future, as we delve into database operations, we'll explore testing more extensively.At this point, it's crucial to appreciate the simplicity of incorporating tests whenever we introduce new features to our Django project.

## Conclusion

In this chapter, we covered the essential components of a Django web application: URLs, views, and templates. Defining URL patterns allows us to map user requests to specific views, the core of our application logic. Configuring views shape the behavior and content presented to users.

Templates are crucial, separating the presentation layer from business logic. We created the base.html template for a consistent structure and crafted specialized templates like home.html and about.html for tailored user experiences.

Integrating these elements enabled a seamless flow of data from views to templates, creating a dynamic web experience. This chapter establishes the foundation for advanced Django topics, encouraging exploration of its extensive capabilities. With a solid grasp of URLs, views, and templates, we can confidently build robust, user-friendly web applications. Subsequent sections will cover advanced features and best practices to enhance our Django development skills.

# Chapter 4: Admin, Models, and Databases

In this chapter, we will continue working on our newspaper_app project. We learned about URLs, views, and templates in the last chapter. In this chapter, we will talk about models, databases, and the admin's access to our website. First, let's create a model.

A model is a Python class that defines the structure and behavior of a database table. Each attribute of the model defines a field in the table, and each method defines the behavior of the table. Models are used to interact with the database and allow it to create, retrieve, update, and delete records. They serve as the high-level abstraction for database operations.

Having said that, let's create our first model.

**news/models.py**

```
# First, we need to import the necessary module from Django.
from django.db import models

# Now, let's define a class named "Post" that inherits from Django's "Model"
class.
class Post(models.Model):
    # Inside our "Post" class, we define a field named "text" of type
"TextField".
    # In Django, a model represents a database table, and a field represents a
column in that table.
    # Here, we're saying that our "Post" model will have a column called "text"
that stores text data.
    text = models.TextField()

    #In the following code snippet, we've added a __str__ method to the Post
model.
    #This method defines how a Post object should be represented as a string.
    #In this case, it displays the first 50 characters of the text field as the
object's string representation.

    def __str__(self):
        return self.text[:50]  # Display the first 50 characters of the text as
the object's string representation
```

Put simply, this code establishes a Django model named "Post," which includes a lone field called "text" for storing textual data. Django will convert the "Post" model into a database table, with the "text" field serving as a column within that table to house the actual content of the post.

To make this model available in the database, you'll need to create and apply migrations. Open your terminal and run the following commands:

***command prompt***

```
> python manage.py makemigrations news
> python manage.py migrate
```

These commands will generate the necessary database schema changes based on the model you defined and apply them to your database. Next, you can create a superuser who can log in to the Django admin interface. Run the following command:

***command prompt***

```
> python manage.py createsuperuser
Username (leave blank to use 'pc'): test_name
Email address: test_email@email.com
Password:
Password (again):
Superuser created successfully.
```

After running this command, you will be prompted to provide a username, email, and password for the superuser. Once you complete this step, the superuser will be created successfully.

You can choose any username, email, and password.

| Note: Password won’t be visible on the screen for security reasons. |
|---|

Also, please remember the password that you have entered. My password is: testuser@111
Once you complete this step, the superuser will be created successfully. Now let’s update our admin.py. Here we will integrate our models into the admin panel so that the admin can work with the database from the admin panel.

**news/admin.py**

```
# First, we are importing the `admin` module from Django, which provides tools
# for managing the Django admin interface.
from django.contrib import admin

# Next, we are importing the `Post` model from the current directory (indicated
```

```
# by the dot `.`) under the `models` module. This assumes that there is a file
# named `models.py` in the same directory as this code, and it contains a
# definition for the `Post` model.
from .models import Post

# Now, we are registering the `Post` model with the Django admin. This means
# that the `Post` model will be visible and manageable through the Django admin
# interface. This is very convenient for performing CRUD (Create, Read, Update,
# Delete) operations on the `Post` model through a web-based interface.
admin.site.register(Post)
```

In this code snippet, we import the necessary modules and register the Post model with the Django admin interface. This allows you to manage and view Post objects through the admin interface.

With these code snippets and explanations, you should have a clearer understanding of how to create a Django model, perform database migrations, create a superuser, and register the model with the admin interface.

Now let's update our views.py.

**news/views.py**

```
# We are importing a class called 'ListView' from the 'django.views.generic'
module. This class is a generic view that helps to display a list of objects.

from django.views.generic import ListView

# We are importing the 'Post' model from the current directory (indicated by the
dot).
from .models import Post

# We are defining a new class called 'HomePageView' that inherits from
'ListView'.
#Inheritance means that HomePageView will have all the functionality of ListView
and can also have additional features or modifications.

class HomePageView(ListView):
    # We are specifying the model that is connected to this view.
    model = Post

    # We are assigning the template file that will be used to render this view.
    template_name = 'index.html'
```

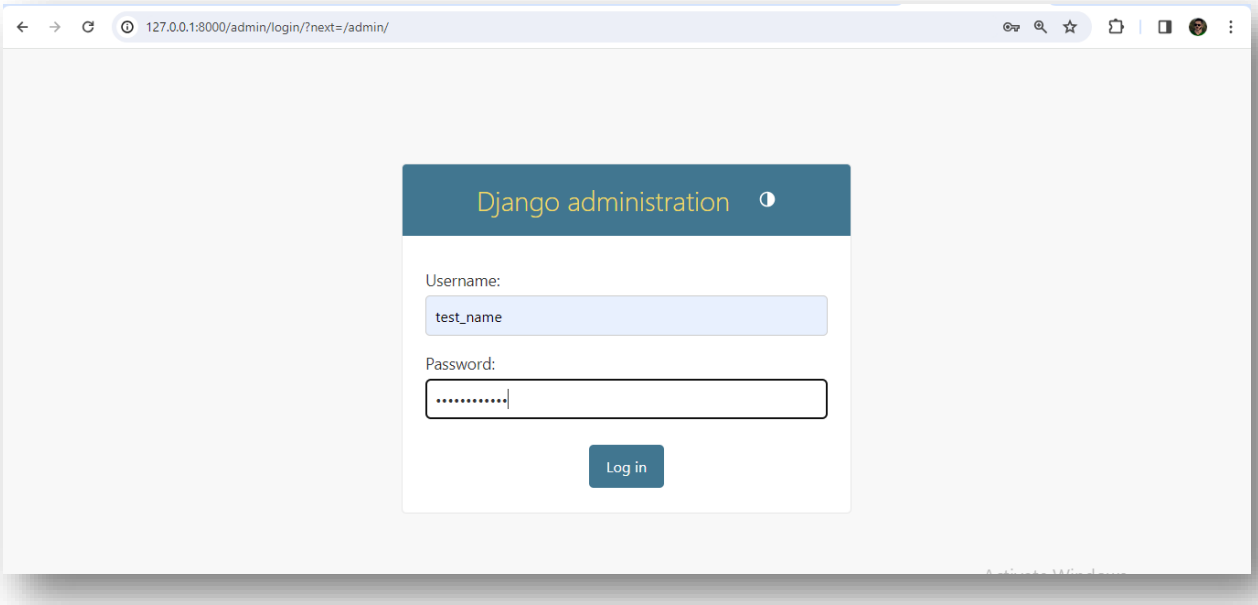
```
class AboutPageView(TemplateView):
    template_name = 'about.html'
```

Now, let's update our template files.

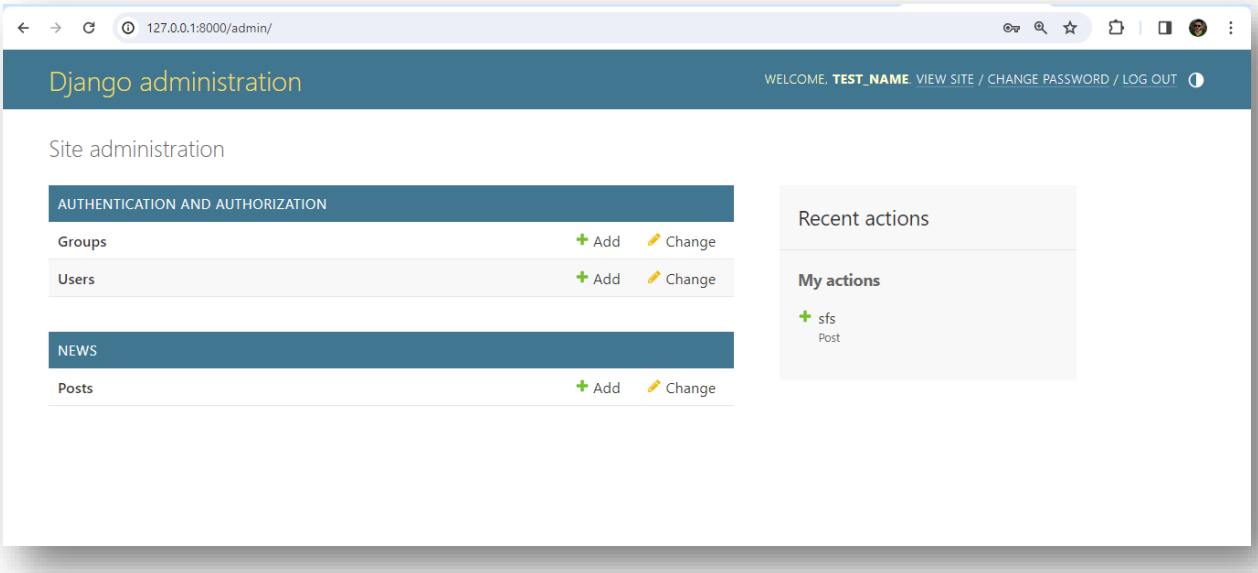
**templates/index.html**

```
<!-- This line indicates that this template extends another template called
'base.html'. -->
{% extends 'base.html' %}

<!-- This block tag is used to define a content block within the extended
template. -->
<!-- In this case, it's defining the 'content' block. -->

{% block content %}
  <!-- Inside the 'content' block, we have an HTML heading tag displaying
"Homepage". -->
  <h1>Homepage</h1>

  <!-- Now, we're using a for loop to iterate over a list of objects, which are
posts. -->
  <!-- The 'object_list' the defalt variable contains a list of post objects. -->

  {% for post in object_list %}
    <!-- Inside the loop, we are displaying the content of each post. -->
    {{ post }}
  {% endfor %}

<!-- This line marks the end of the 'content' block. -->
{% endblock content %}
```

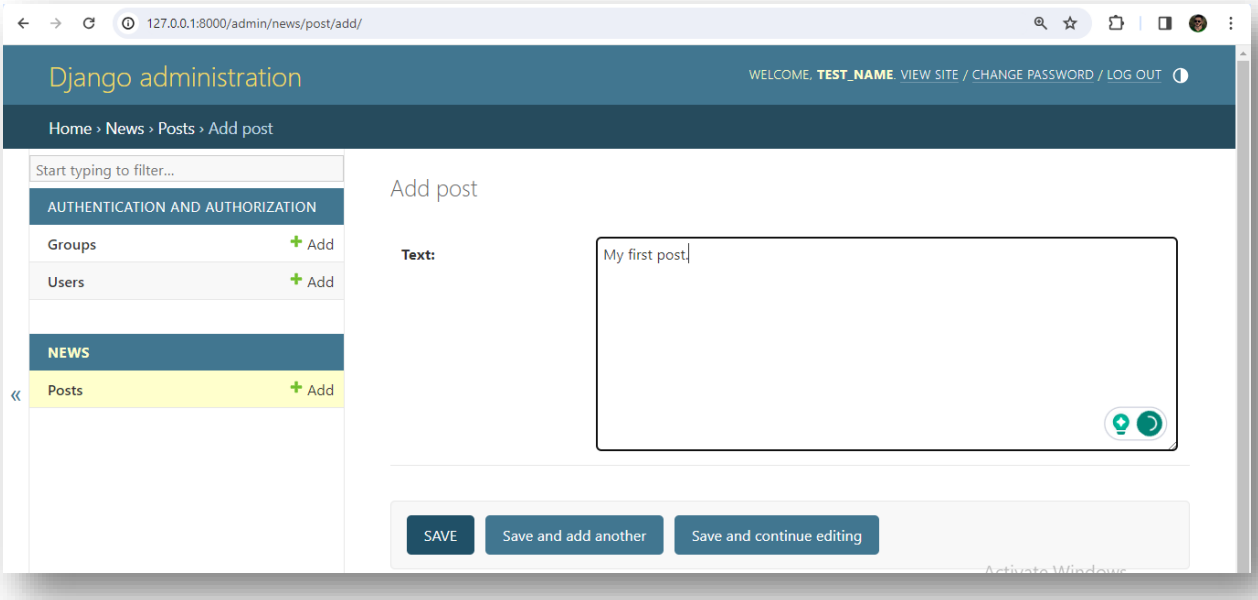
Now that we have updated all the necessary files, it's time to see our admin panel.
Visit: http://127.0.0.1:8000/admin/and sign in as the superuser..




This is how our admin panel will look like once you are logged in as the superuser or admin.You will see a posts row in the news section there.




Click on Add button.




Write your content for your post and click on save. Now refresh the homepage.


Go back to the admin panel. Add a few more posts and see how it looks.


Now, this is one way of displaying our post, or we can customize it. Go to the views.py and update the file with the following code.

**news/views.py**

```
from django.shortcuts import render

from django.views.generic import TemplateView

from django.views.generic import ListView
from .models import Post
class HomePageView(ListView):
    model = Post
    template_name = 'index.html'

    # Defining the name of the variable that will be used to pass the list of
posts to the template
    context_object_name = 'all_posts_list'

class AboutPageView(TemplateView):
    template_name = 'about.html'
```

**context_object_name** is an attribute provided by Django's class-based views. It is an attribute that you can set when defining a class-based view to specify the variable name that will be used in the template to represent the list of objects.

Here's how it works:

In the context of a Django class-based view, the ListView (and other similar views) automatically retrieves a list of objects from a specified model and passes it to the template for rendering. The context_object_name attribute allows you to control the name of the variable that will hold this list of objects in the template's context.

| By default, the name is **object_list**. |
|---|

In this specific view, we are naming it **'all_posts_list'**. You can choose any other name if you want. Now we have to update our index.html template file accordingly.

**templates/index.html**

```
{% extends 'base.html' %}
{% block content %}
  <h1>Homepage</h1>


  {% for post in all_posts_list %}
  {{ post }}
  {% endfor %}


{% endblock content %}
```

And with these changes, you will see the homepage in the same way as before.

## Tests

Now let's create tests for our updated application since we have created models in our app.

**news/tests.py**

```
from django.test import TestCase
from django.urls import reverse
from .models import Post

class HomePageViewTest(TestCase):
    def setUp(self):
```

```
        self.post1 = Post.objects.create(text='Test Post 1')
        self.post2 = Post.objects.create(text='Test Post 2')
        self.url = reverse('home')

    def test_home_page_status_code(self):
        response = self.client.get(self.url)
        self.assertEqual(response.status_code, 200)

    def test_home_page_uses_correct_template(self):
        response = self.client.get(self.url)
        self.assertTemplateUsed(response, 'index.html')

    def test_home_page_displays_posts(self):
        response = self.client.get(self.url)
        self.assertContains(response, self.post1.text)
        self.assertContains(response, self.post2.text)

    def test_home_page_uses_correct_context_object_name(self):
        response = self.client.get(self.url)
        self.assertTrue(response.context['all_posts_list'])

class PostModelTest(TestCase):
    def setUp(self):
        self.post = Post.objects.create(text='Test Post Content')

    def test_text_content(self):
        post = Post.objects.get(id=1)
        expected_object_name = f'{post.text[:50]}'
        self.assertEquals(expected_object_name, str(post))
```

This is a basic set of tests to check the status code, template used, whether posts are displayed, and if the correct context object is used.

The second test checks whether the text field of the Post model is properly represented as a string. It retrieves the post created in the setUp method and compares its string representation to the expected value based on the first 50 characters of the text field.

To execute the tests, stop the server using Control+c, and enter the command "python manage.py test" in the command line:

***command prompt***

```
> python manage.py test
```

You will get a similar output as the following:

**Output:**

***command prompt***

```
Found 5 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
----------------------------------------------------------------------
Ran 5 tests in 0.066s

OK
Destroying test database for alias 'default'...
```

## Conclusion

This chapter has delved into the fundamental pillars of Django development: Admin, Models, and Databases.
We explored the robust Django Admin interface, empowering developers to effortlessly manage and manipulate data through a user-friendly interface.

The pivotal role of Models in defining the data structure and behavior of our applications was thoroughly examined, emphasizing the importance of precision and coherence in crafting model classes.

Additionally, we navigated the intricacies of databases, understanding how Django seamlessly abstracts the underlying database system while providing powerful tools for data retrieval and manipulation.

As we continue our journey through Django, the insights gained in this chapter will serve as a solid foundation for constructing scalable, maintainable, and efficient web applications.

# Chapter 5: Working with Static Files

In the dynamic realm of web development, creating visually appealing and user-friendly websites is a top priority. A crucial component in achieving this is managing static files.

Static files, such as images, stylesheets, JavaScript, and more, are the foundation upon which a website's aesthetics and interactivity are built.

So, In this chapter, will talk about static. While Django excels at handling dynamic content, it also provides a robust infrastructure for organizing, serving, and optimizing static files. Whether you're developing a simple blog or a complex web application, understanding how to effectively manage and utilize static files is essential for creating a polished and engaging user experience.

We will explore the intricate process of configuring Django to handle static files efficiently, taking advantage of various tools and techniques.

From setting up a structured directory layout to optimizing file delivery for performance, this chapter provides comprehensive guidance for web developers at all levels.

By the end of this chapter, you'll be well-versed in the art of managing static files in Django, enabling you to enhance your web applications with beautiful designs, seamless interactivity, and an overall superior user experience. Let's dive into the world of static files and unleash their potential in your Django projects.

Let’s create a new project; you already know the process. Open your command prompt and choose the folder where you want to create your project.

***command prompt***

```
> cd Desktop\django_projects
```

Create the folder and name your project:

The following command is used to create a new directory (folder) named "**Advanced_newspaper_app**" in the current directory.

***command prompt***

```
>mkdir Advanced_newspaper_app
```

The following command changes the current working directory to " **Advanced_newspaper_app**" After running this command, you will be working within the " **Advanced_newspaper_app** " directory.

*command prompt*

```
>cd Advanced_newspaper_app
```

The following command uses pipenv to create a Python virtual environment and install Django within that virtual environment. As you know pipenv is a tool used to manage Python dependencies and create isolated development environments. In this case, it installs Django.

*command prompt*

```
>pipenv install django
```

The following command activates the virtual environment created in the previous step. Activating the virtual environment ensures that when you run Python or Django-related commands, they will use the dependencies installed within the virtual environment rather than the system-wide Python environment. It isolates your project's dependencies from other projects and the system.

*command prompt*

```
>pipenv shell
```

Now, let's create a new Django project called " **Advanced_newspaper_app_project** " in the current directory (indicated by the dot .).

*command prompt*

```
>django-admin startproject Advanced_newspaper_app_project .
```

"**Advanced_newspaper_app_project** " will be the root directory of your Django project, and it will contain the necessary files and settings for your project.

Now, let's create a new Django app named "news" within your project. Django apps are modular components of a Django project, allowing you to organize and separate different parts of your application.

*command prompt*

```
>python manage.py startapp news
```

The migrate command is used to apply any pending database migrations. In Django, migrations are a way to track and propagate changes to the database schema. This command ensures that the database structure is in sync with your project's models.

*command prompt*

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

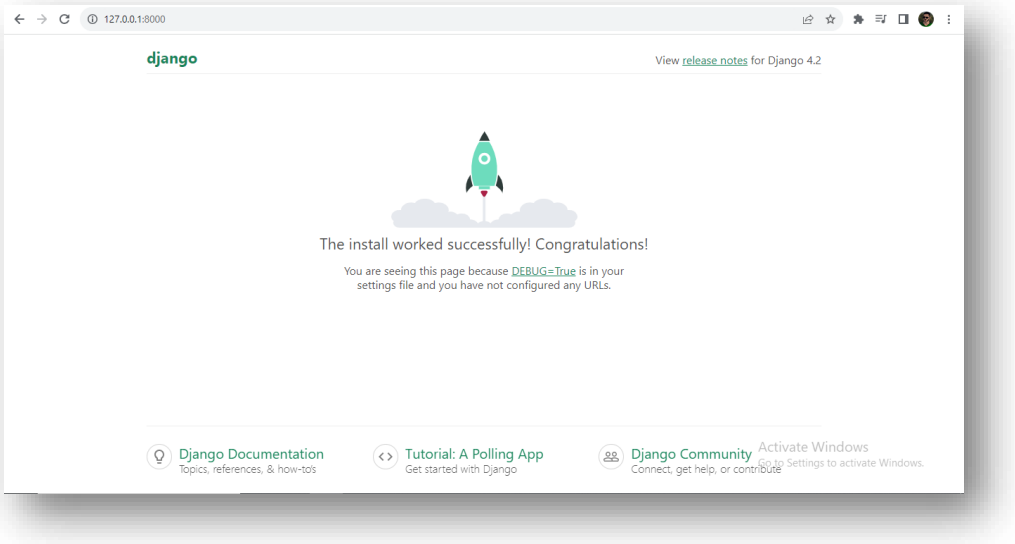
After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

After running this command, you should see an output indicating that the development server is running. By default, it listens on port 8000.

Or you can go to the link http://127.0.0.1:8000/ in your browser.



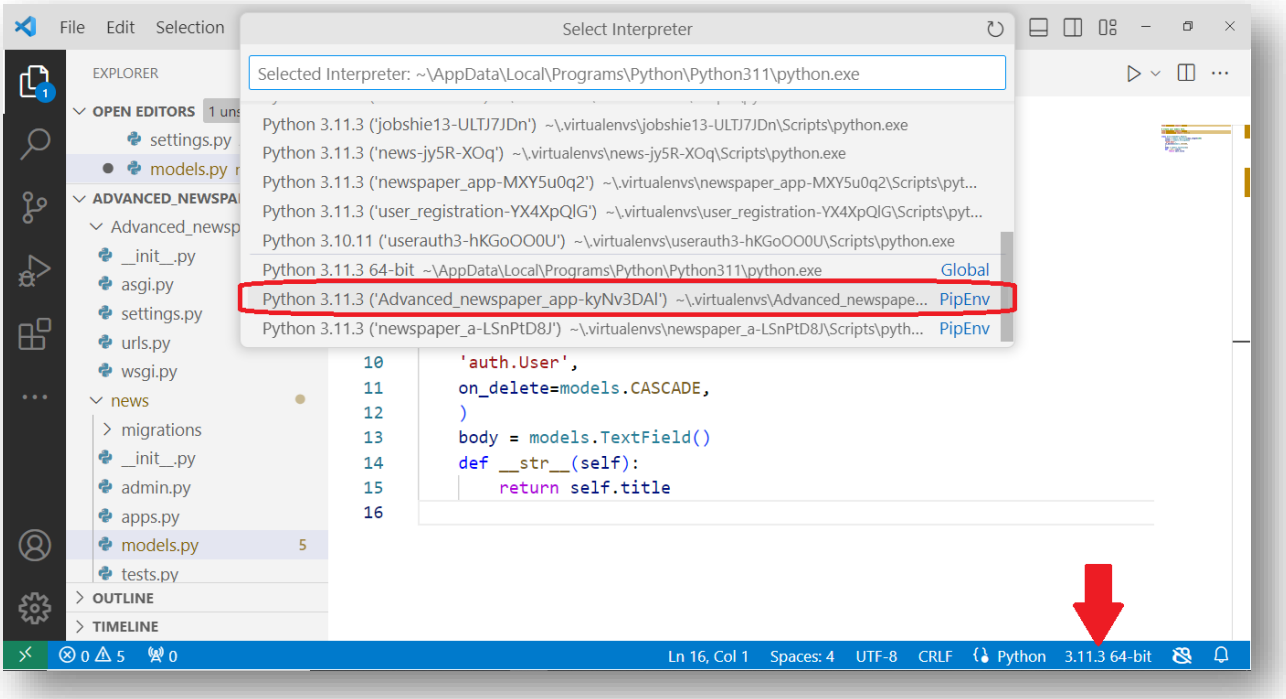

To inform Django about our newly created app, launch our text editor and insert the app into the INSTALLED_APPS list within our settings.py file.

**Settings.py**

```
INSTALLED_APPS = [

    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'news',
```

```
]
```

Now let's select the correct interpreter. You can do that by pressing **Ctrl+Shift+P** or clicking on the point where the red arrow is pointing in the following image.




Now, let's create our model:

**news/models.py**

```
# First, we're importing necessary modules from Django.
from django.db import models

# Now, we define a Django model named Article.
# In Django, a model represents a database table.
class Article(models.Model):
    # This is a field for the title of the article.
    # CharField is used for short to medium text (like a title), and max_length
is the maximum length of the title.
    title = models.CharField(max_length=200)

    # Here, we're creating a foreign key relationship to the User model from the
'auth' app.
    # A ForeignKey is a link from one model to another.
    # This is saying that each article has an author, and the author is a user.
```

```
    # on_delete=models.CASCADE means that if a user is deleted, all their
articles will be deleted as well.
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE,
    )

    # This is a field for the body of the article.
    # It's a TextField, which is suitable for longer text like the body of an
article.
    body = models.TextField()

    # This method provides a string representation of the object.
    # In this case, it returns the title of the article.
    def __str__(self):
        return self.title
```

In summary, this code defines a Django model called Article with three fields: title (for the title of the article), author (a ForeignKey linking to the User model for the author of the article), and body (for the content of the article). Additionally, there's a method __str__ which returns the title of the article when the object is printed or displayed. This model can be used to create, retrieve, update, and delete articles in a Django application.

But we have introduced a new term here 'auth.User'. Let's understand what this is.

**'auth.User':**

- This refers to the User model provided by Django in the auth app.
- The User model is a built-in model that represents a user account, and it includes fields like username, password, email, etc.
- In the context of the Article model, models.ForeignKey('auth.User', on_delete=models.CASCADE) specifies a relationship between the Article model and the User model. It means that each article is associated with a single user (author), and if that user is deleted, all their associated articles should be deleted as well (on_delete=models.CASCADE).

So, 'auth.User' is simply the way to reference the User model in Django when you're creating relationships between models. It's a string that includes the app name (auth) and the model name (User).

***command prompt***

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

This command is used to create migration files for a Django app. Migration files are a way to track and store changes to your database schema over time. When you make changes to your models, you create migration files to record those changes. These migration files are stored in the "migrations" directory of the app.

***command prompt***

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

This command is used to apply the pending database migrations to the database. It takes the migration files generated in the previous step and executes the SQL statements necessary to bring the database schema in sync with the current state of the models.

## Let's create a super user

***command prompt***

```
>python manage.py createsuperuser
Username (leave blank to use 'pc'): su1
Email address: su1@email.com
Password:
Password (again):
Superuser created successfully
```

**Superuser created successfully:** If you've successfully provided all the required information, Django will generate a superuser account with the given username, email, and password. This message confirms that the superuser has been created.

In the example, a superuser with the username "su1," the email address "su1@email.com," and a password (not shown) has been successfully created. This superuser will have elevated privileges within the Django application, allowing them to access and manage administrative features.

Now let's update our admin.py

**news/admin.py**

```python
from django.contrib import admin
from .models import Article
admin.site.register(Article)
```

This code sets up the Django admin interface to manage the Article model, making it easier to interact with and manage data related to articles in our web application.

Now let's create a new urls.py for the news app.

*command prompt*

```
>echo> news/urls.py
```

Now let’s add the URL to our homepage.

**news/urls.py**

```
from django.urls import path
from .views import ArticleListView
urlpatterns = [
path('', ArticleListView.as_view(), name='home'),
]
```

This code sets up a URL pattern for the home page of our Django web application, linking it to the ArticleListView class and giving it the name 'home'. Now let’s update the URL file for our project. It imports necessary modules: admin and path from django.contrib, and include from django.urls.

The urlpatterns list is a list of URL patterns defined for this Django project. Each URL pattern specifies a URL path and the view or application that should handle requests to that path.

The first pattern, path('admin/', admin.site.urls), routes requests with a URL that starts with 'admin/' to the Django admin site. This allows administrators to manage the site through the Django admin interface.

The second pattern, path('', include('blog.urls')), is a bit more dynamic. It routes requests with an empty URL path to the 'blog.urls' module using the include function. This means that any URL without a specific path after the domain name (e.g., 'example.com/') will be passed to the 'blog.urls' module for further processing. The include function is often used to include additional URL configurations defined in other modules, enabling you to structure your project's URL handling in a modular way.

**advanced_newspaper_app_project/urls.py**

```
# First, we're importing necessary modules from Django.
# 'admin' is for the Django admin interface, and 'path' and 'include' are for URL
routing.
from django.contrib import admin
from django.urls import path, include

# Now, we define our URL patterns using a list called 'urlpatterns'.
# This list will determine how our web application responds to different URLs.

urlpatterns = [

# The first pattern maps the 'admin/' URL to the Django admin interface.
```

```
# So, when someone goes to 'yourdomain.com/admin/', they'll access the admin
panel.
path('admin/', admin.site.urls),

# The second pattern is an empty string ('') mapped to the 'news.urls' module.
# This means that any URL that doesn't match 'admin/' will be passed to the
'news.urls' module.
# The 'include' function is used to include the URL patterns defined in
'news.urls'.
# It allows us to organize our URL patterns in a modular way.
path('', include('news.urls')),

]

# That's it! These URL patterns tell Django where to direct users based on the
URL they request.
# You'll find the detailed URL configurations inside the 'news.urls' module.
# This file just serves as the entry point for the URLs of your Django project.
```

Let’s create a new directory to handle our templates.

***command prompt***

```
>mkdir templates
```

This command creates a directory (folder) named "templates" in the current directory. It's used for organizing files in a structured way.

***command prompt***

```
>echo> templates/base.html
```

This command uses the echo command to create a new file named "base.html" inside the "templates" directory.

***command prompt***

```
>echo> templates/home.html
```

Similar to the previous command, this creates an empty "home.html" file inside the "templates" directory. Now, let’s update our views.py file.

**news/views.py**

```
# Importing necessary components from Django
from django.views.generic import ListView
```

```
# Importing the Article model from the current directory
from .models import Article

# Creating a new class named ArticleListView, which inherits from Django's
ListView
class ArticleListView(ListView):
    # Specifying the model to be used, which is the Article model we imported
    model = Article

    # Specifying the template to be used, in this case, 'home.html'
    template_name = 'home.html'
```

This code defines a class ArticleListView that utilizes Django's generic ListView to display a list of articles using the Article model and a template named home.html. This is a common pattern in Django for displaying lists of items in a web application.

Now, let's define the path to our templates file in settings.py.

***settings.py***

```
from pathlib import Path
import os
```

**import os**: Import the Python os module at the beginning of your settings.py file. The os module is essential for handling operating system-related tasks. Placing the import statement at the top of the file ensures that all subsequent code can make use of the os module.

Now, let's update the TEMPLATES section in settings.py.Here's the relevant portion of the code:

***settings.py***

```
TEMPLATES = [
    {

        # Other templates settings...

        'DIRS': [os.path.join(BASE_DIR, 'templates')],   # This line specifies
the template directory.

        # Other templates settings...
  },
]
```

Now let's update the base.html file.

***temlates/base.html***

```
{% load static %}
<!-- This line is used to load the static files like CSS and JavaScript.
     It's a Django template tag that tells the template engine to include
     the necessary static files for the page. -->

<html>
<head>
    <title>Advanced Newspaper App</title>
</head>
<body>
    <header>
        <div>
            <!-- This is the header section of our web page. -->
            <h1><a href="{% url 'home' %}"> Advanced Newspaper App </a></h1>
            <!-- This is a link to the home page. The {% url 'home' %}
                 template tag is used to generate the URL for the 'home'
                 view in our Django application. -->
        </div>
    </header>

    <div>
        <!-- This is the main content section of our web page. -->
        {% block content %}
        <!-- The {% block content %} and {% endblock content %} tags are
             used to define a block named 'content' that can be overridden
             in child templates. The actual content for this block will be
             provided in other templates that extend this one. -->
        {% endblock content %}
    </div>
</body>
</html>
```

In summary, this is a basic HTML template for a web page. It includes a header with a link to the home page and a content section that can be customized by child templates. The {% load static %} tag is used to load static files, and the {% url 'home' %} tag generates the URL for the 'home' view in the Django application.

Now let’s update the home.html

***temlates/home.html***

```
{% extends 'base.html' %}
<!-- This line indicates that this template extends another template called
'base.html'.
     It's a way to reuse common elements across multiple pages in your Django
project.
     The content of this template will be placed in a block defined in
'base.html'. -->

{% block content %}
<!-- Here, we define a block named 'content'.
     This is a placeholder where the content specific to this template will be
inserted.
     In 'base.html', you'd typically find something like {% block content %}{%
endblock %}. -->

{% for article in object_list %}
<!-- Now, we're using a for loop to iterate through each 'article' in the
'object_list'.
     'object_list' is a list of articles, and we're looping through each one. -->

<div class="post-entry">
<!-- For each article, we create a <div> element with a class 'post-entry'.
     This is a common practice in web development to style elements with CSS. We
will talk about the css file later in this chapter. -->

<h2><a>{{ article.title }}</a></h2>
<!-- This will display the title of the article as a clickable link.
     '{{ article.title }}' is Django's template syntax to output the title
attribute of the 'article' object. -->

<p>{{ article.body }}</p>
<!-- '{{ article.body }}' is again using Django's template syntax to output the
body attribute of the 'article' object. -->

</div>
<!-- We close the <div> for each article. This helps in maintaining the structure
of our HTML. -->

{% endfor %}
<!-- This marks the end of the for loop. -->

{% endblock content %}
<!-- This marks the end of the 'content' block.
```

```
    The content inside this block will be placed where the corresponding block
is defined in 'base.html'. -->
```

In summary, this code is a Django template that extends another template ('base.html'). It defines a block called 'content' and uses a for loop to iterate through a list of articles. For each article, it creates a  with a title and body displayed inside it. The structure and styling are handled in the 'base.html' template, providing a consistent look across multiple pages.

After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

Visit the link http://127.0.0.1:8000/ in your browser.


**The Static Files**

Now, let's create a new folder to handle our static files.

***command prompt***

```
> mkdir static
```

Next, specify the path to our static folder inside the `settings.py`.

***settings.py***

```
# Static files include things like CSS, JavaScript, and images that don't change
dynamically.

# The STATIC_URL is the URL prefix for static files.
# When you reference static files in your templates or code, this prefix is used.
# In this case, 'static/' means that static files will be served from a URL like
'/static/'.

STATIC_URL = 'static/' # You will find this line already written in settings.py
file.

# STATICFILES_DIRS is a list of directories where Django will look for static
files.
# These are additional locations, apart from the app-specific 'static'
directories.
# Here, we use os.path.join to join the project's base directory (BASE_DIR) with
'static'.
# It tells Django to also look for static files in the 'static' directory at the
project level.

STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

Now, let's create our first static file called base.css inside the static folder.

***command prompt***

```
> echo> static/base.css
```

Now let's update the base.css to make the header in the red color.

***static/base.css***

```
    header h1 a {
    color: red;
    }
```

Now we have to specify the path to css file in the base.html file. This will help in maintaining the same style throughout the web application.

***templates/base.html***

```
{% load static %}
<html>
<head>
```

```
<title>Advanced Newspaper App</title>

<!-- Linking to a static CSS file named 'base.css' for styling. -->

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

</head>
```

After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

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

Visit the link http://127.0.0.1:8000/.


**Integrating Google fonts in our web design:** One cool thing we can do is use Google style sheets in our web application.

**Access Google Fonts:** We can Include a link to the Google Fonts stylesheet in our HTML file to fetch font files from Google's servers.

**HTML Link Inclusion:** We can use the HTML <link> tag to insert the Google Fonts stylesheet, specifying the URL with the href attribute and defining it as a stylesheet using the rel attribute.

***templates/base.html***

```
{% load static %}
<html>
<head>
<title> Advanced Newspaper App </title>

<link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400"
rel="stylesheet">

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

We can customize the font name and weight in the URL to select different fonts and variations from the Google Fonts library.

Dynamic Loading: As our web page loads, the browser requests the font files from Google's servers using the link to the Google Fonts stylesheet.

Applying Fonts: Once downloaded, the browser applies these fonts to HTML elements on our page. We can use the font-family property in your CSS styles to leverage these fonts.

Now let's style our web app with more CSS code. We mainly focus on Django and Python in this book, so we will not discuss the CSS code in detail.

***static/base.css***

```
body {
    font-family: 'Source Sans Pro', sans-serif;
    font-size: 18px;
    }
    header {
    border-bottom: 1px solid #999;
    margin-bottom: 2rem;
    display: flex;
    }
    header h1 a {
    color: red;

    text-decoration: none;
    }
    .nav-left {
    margin-right: auto;
    }
    .nav-right {
    display: flex;
    padding-top: 2rem;
```

```
    }
    .post-entry {
    margin-bottom: 2rem;
    }
    .post-entry h2 {
    margin: 0.5rem 0;
    }
    .post-entry h2 a,
    .post-entry h2 a:visited {
    color: blue;
    text-decoration: none;
    }
    .post-entry p {
    margin: 0;
    font-weight: 400;

    }
    .post-entry h2 a:hover {
    color: red;
    }
```

For now, we have only one page, which is the homepage. On the homepage, we can list all the titles of the articles, but for now, we don't have a page to show the content (body) of that article.

Now, let's create a page to display the content (body) of an article. For that, we have to create a view to display the body of an article.

***news/news.py***

```
# Importing necessary classes from Django
from django.views.generic import ListView, DetailView
from .models import Article

class ArticleListView(ListView):
    model = Article
    template_name = 'home.html'

# Defining a class for a detailed view of a single article
class ArticleDetailView(DetailView):
    # Again, specifying the model (Article)
    model = Article
    # Specifying the template for rendering the detailed view
(article_detail.html)
    template_name = 'article_detail.html'
```

In summary, this code sets up two views for handling lists of articles and detailed views of individual articles using Django's generic views.

Now let’s create the article_detail.html file.

***command prompt***

```
>echo> templates/article_detail.html
```

Now let’s update the article_detail.html file.

***templates/article_detail.html***

```
 {% extends 'base.html' %}
{% block content %}
<div class="post-entry">
<h2>{{ object.title }}</h2>
<p>{{ object.body }}</p>
</div>
{% endblock content %}
```

Note that we are not utilizing {{ article.title }} or {{ article.body }} tags in the article_detail.html file.

The choice between {{ object.title }} and {{ article.title }} depends on how your data is structured and passed to the template. If you have a single object in the context, you might use {{ object.title }}. If you have a list of objects and you're iterating over them, you would typically use {{ article.title }} within a loop.

On the homepage, we use {{ article.title }} because we are displaying all the articles there, whereas we use {{ object.title }} on article_detail.html because we are displaying the content of only one specific article.

Don't worry if it seems a little confusing; it will become clearer when you visit the actual web pages of our web application. Now let’s update the news/urls.py and include the URL for the article_detail page.

***news/urls.py***

```
# Importing necessary functions from Django for handling URLs
from django.urls import path

# Importing views (ArticleListView and ArticleDetailView) from the current
package
from .views import ArticleListView, ArticleDetailView

# Defining URL patterns for the Django app
```

```
urlpatterns = [

    # This path is for displaying the details of a specific article.
    # It expects an integer (int:pk) as part of the URL, representing the primary
key
    # of the article in the database. The URL pattern is named 'article_detail'.
    # When this URL is accessed, it will use the ArticleDetailView to handle the
request.
    path('article/<int:pk>/', ArticleDetailView.as_view(),
name='article_detail'),

    path('', ArticleListView.as_view(), name='home'),
]
```

**The primary key**

The primary key, often abbreviated as PK, is a unique identifier assigned to each record in a database table. It serves as a way to uniquely distinguish one record from another. In the context of Django models (which represent database tables), the primary key is automatically created for each model unless specified otherwise.

In the provided Django code, the path('article/int:pk/', ...) URL pattern expects an integer (int:pk) to be included in the URL. This integer corresponds to the primary key of a specific article in the database. When a user accesses a URL like /article/1/, Django will capture the integer 1 as the pk (primary key) parameter. This allows the application to identify and display the details of the article with the primary key of 1.

So, the primary key is used to uniquely identify and retrieve specific records from the database, ensuring that each record can be accessed and manipulated individually. In this case, it helps to display the details of a specific article when its URL is accessed.

Now let's update our homepage.

***templates/home.html***

```
{% extends 'base.html' %}
{% block content %}
{% for article in object_list %}
<div class="post-entry">
<!-- In the follwing code,the href attribute contains a dynamic URL generated by
the {% url %} template tag. It points to the 'article_detail' view and includes
the primary key (article.pk) of the current article as a parameter.-->
```

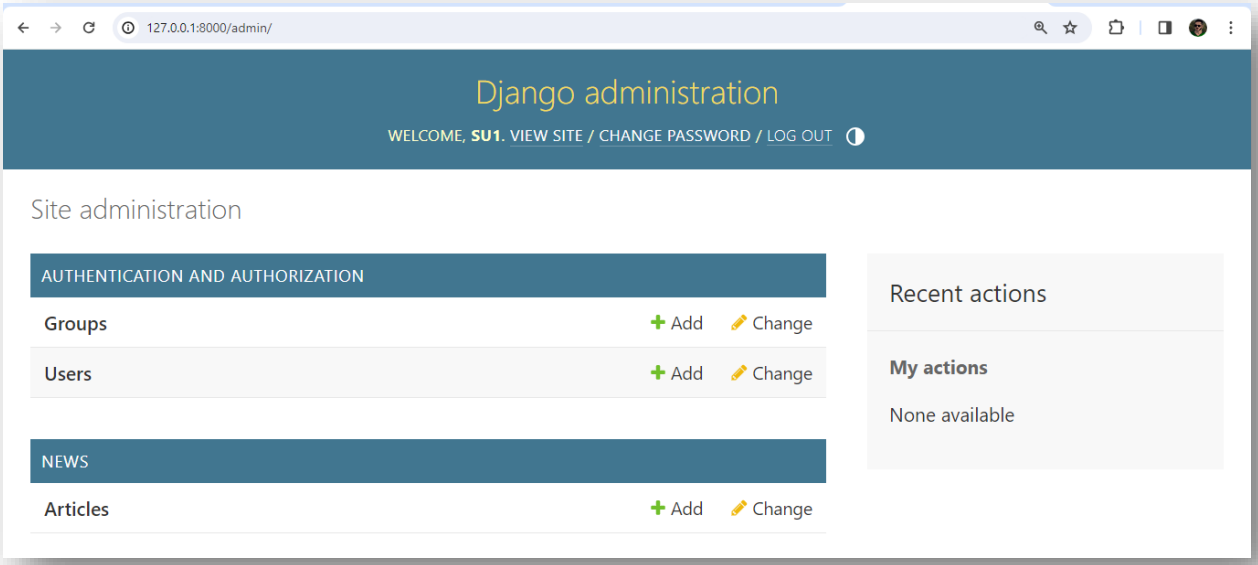
```
<h2><a href="{% url 'article_detail' article.pk %}">{{ article.title }}</a></h2>
<p>{{ article.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

Now, let's see how our homepage looks. After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command 'python manage.py runserver' in the command line.

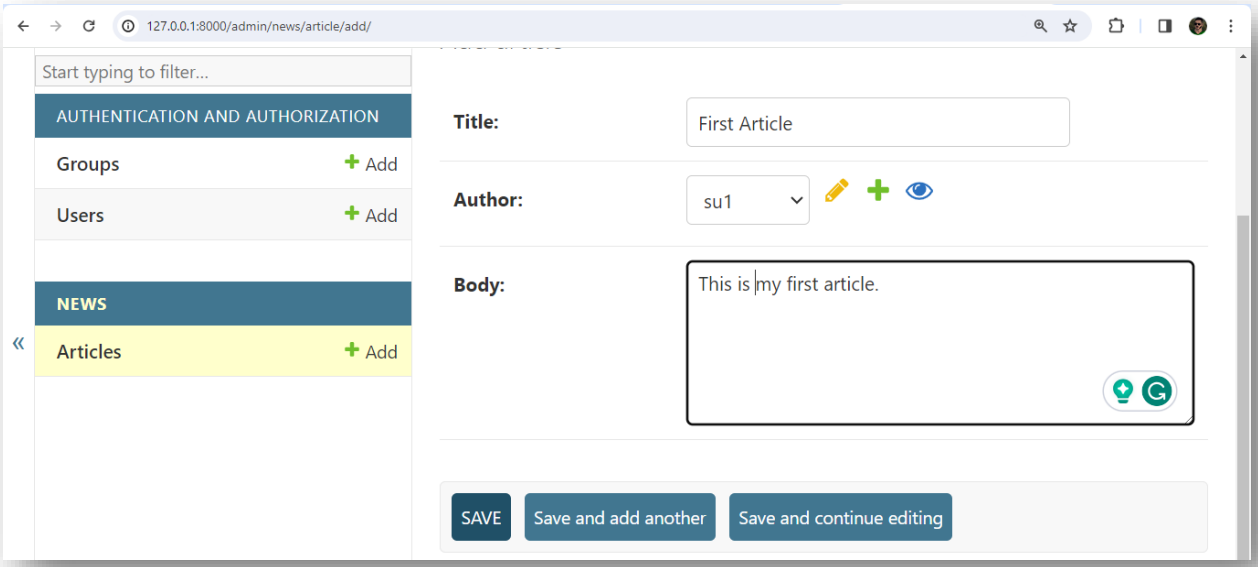
***command prompt***

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

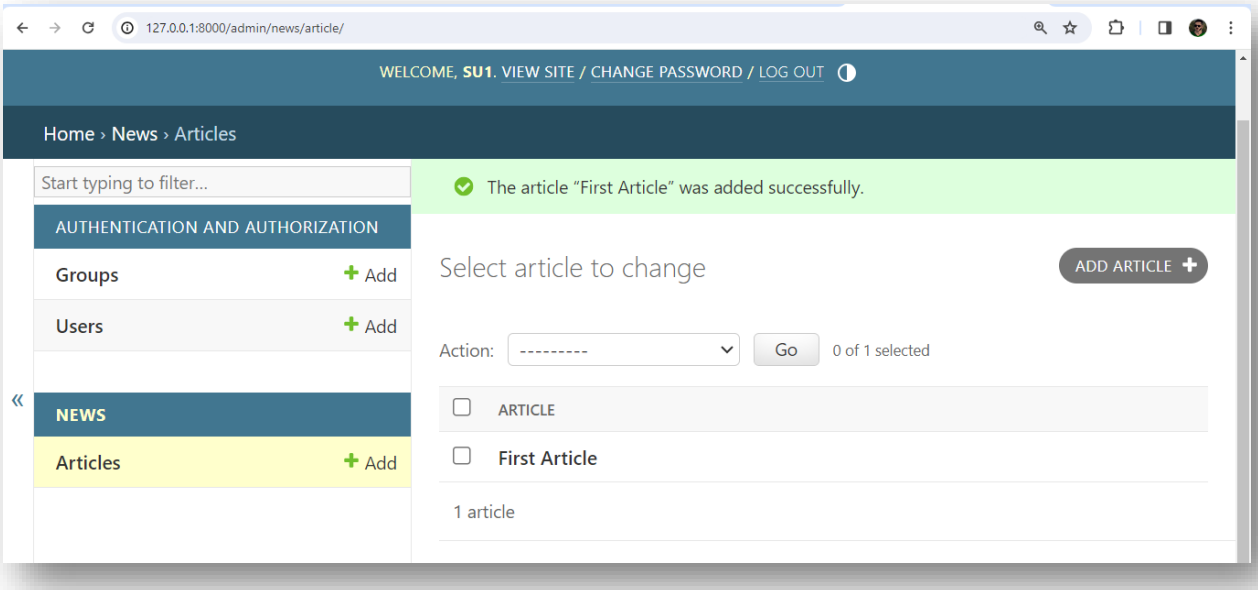

You can see the customization we did through our static files.Now, let’s add some articles to our website. For that, visit: http://127.0.0.1:8000/admin/ and sign in as a superuser. You will see something like this:



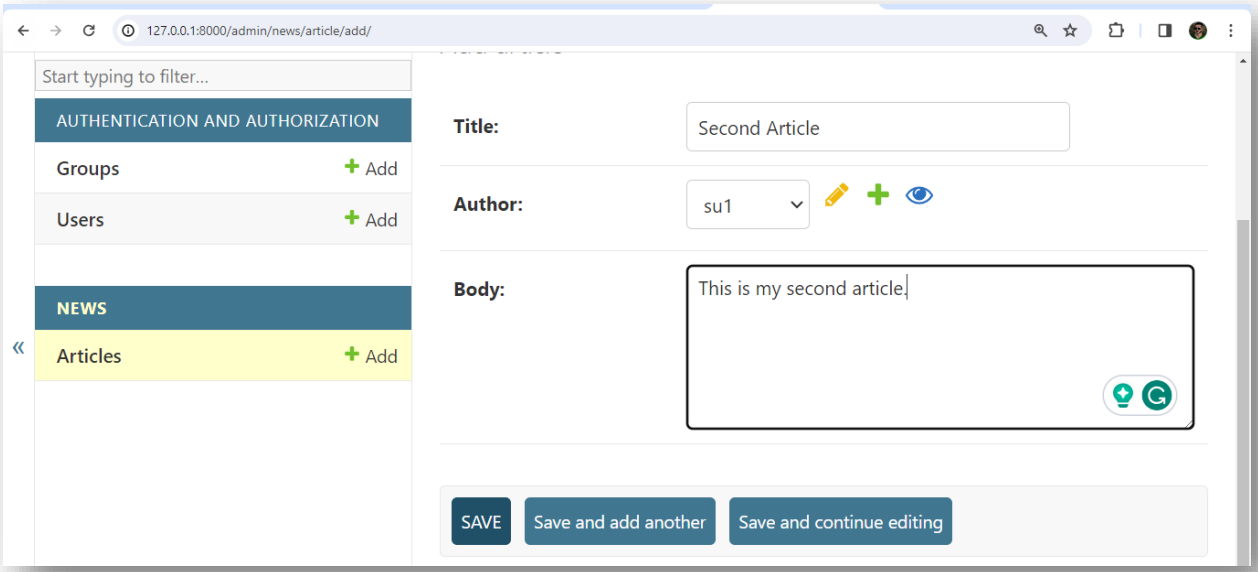

Now click on the "Add" button in the Articles column in the news section. Now write an article and click on save.




You can see your article title there.




Now add one more article and save it.




Now visit the home page and see that your post is visible there with all the customization.


Now click on any article, and you will be able to see the details of that article.

## Conclusion

In summary, a proficient command of static file management is indispensable for crafting dynamic and visually captivating web applications using Django. This section has furnished you with both the understanding and tools necessary for adeptly overseeing static assets, and facilitating the smooth integration of stylesheets into your projects. With a grasp of the intricacies involved in Django's static file handling, you are now well-equipped to elevate user experience and enhance the overall aesthetic appeal of your web applications. As you progress, bear in mind these foundational skills, as they will undoubtedly prove pivotal in the development and success of your Django projects.

# Chapter 6: Forms and User Input

Up until now, we have learned how we can create articles from the admin panel and list them on the homepage. In this chapter, we will take one step forward. In this chapter, we will learn how we can create, update, and delete an article from the main website. We will be able to do all this without using the admin panel. That means we will allow user input using forms.

## Create

Now let's create a "New Article" button.

***templates/base.html***

```
{% load static %}
<html>
<head>
<title> Advanced Newspaper App </title>
<link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400"
rel="stylesheet">

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

<body>
<header>
    <div>
        <h1><a href="{% url 'home' %}"> Advanced Newspaper App </a></h1>

<!-- The following <a> tag creates a link to the 'article_new' view using the {%
url 'article_new' %} template tag.This link allows users to navigate to the page
where they can create a new article.-->

        <a href="{% url 'article_new' %}">+ New Article</a>
        </div>
</header>

<div>
{% block content %}
{% endblock content %}
</div>
</body>
</html>
```

***news/urls.py***

```
from django.urls import path

# Here we import the 'ArticleCreateView' class from the 'views' module of the
current package (denoted by the dot).
from .views import ArticleListView, ArticleDetailView, ArticleCreateView

urlpatterns = [
    # Here, we use the 'path' function to map a URL pattern to a view.
    # The first argument is the URL path, which is 'article/new/' in this case.
    # The second argument is the view that will handle this URL pattern, and it's
'ArticleCreateView.as_view()'.
    # The third argument is the name for this URL pattern, which is
'article_new'.
    path('article/new/', ArticleCreateView.as_view(), name='article_new'),
    path('article/<int:pk>/', ArticleDetailView.as_view(),
name='article_detail'),
    path('', ArticleListView.as_view(), name='home'),
]
```

In summary, the new code sets up a URL pattern for creating new articles. When a user accesses the 'article/new/' URL, Django will route the request to the ArticleCreateView class, presumably handling the creation of a new article. The name 'article_new' can be used in the code or templates to reference this specific URL pattern.

These are just a few examples of URL patterns that you can use in a Django URLconf file. You can create as many or as few URL patterns as you need to handle the different types of requests that your application will receive. Now let's update our views.py.

***news/views.py***

```
from django.views.generic import ListView, DetailView
from .models import Article
# Here we're importing a specific class called CreateView from the
django.views.generic.edit module.
from django.views.generic.edit import CreateView


class ArticleListView(ListView):
    model = Article
    template_name = 'home.html'
```

```
class ArticleDetailView(DetailView):
    model = Article
    template_name = 'article_detail.html'

# Now, we're defining a new class called ArticleCreateView, and it inherits from
CreateView.
class ArticleCreateView(CreateView):
    # Inside our class, we set a few attributes to configure the behavior of our
view.

    # 'model' specifies the model that this view is associated with.
    # In this case, it's a model named 'Article'. This implies that the view will
be used to create new instances of the Article model.
    model = Article

    # 'template_name' is the name of the template file that will be used to
render the HTML for this view.
    # Here, it's set to 'article_new.html', so Django will look for a template
with that name when rendering the view.
    template_name = 'article_new.html'

    # 'fields' is a list of fields from the model that should be included in the
form.
    # The '__all__' value means that all fields from the Article model will be
included.
    fields = '__all__'

# That's it!  we've created a view that can be used to create new Article
instances.
```

This code is essentially creating a form view for creating new articles using Django's class-based views. The CreateView is a generic view provided by Django that simplifies the creation of forms for generating new instances of a model. In this case, it's associated with the Article model, uses the 'article_new.html' template, and includes all fields from the Article model in the form.

Now, let's create the 'article_new.html' file.

***command prompt***

```
>echo> templates/article_new.html
```

Now let’s update the newly created file.

***templates/article_new.html***

```
{% extends 'base.html' %}

{% block content %}
```

```
    <h1>New article</h1>
    <!-- This is a simple HTML heading (h1) displaying "New article". -->

    <form action="" method="post">
        <!-- This is an HTML form element. The 'action' attribute is empty,
indicating that the form will be submitted to the same URL it originated from.
The 'method' attribute is set to "post", indicating that form data will be sent
to the server as an HTTP POST request. -->

        {% csrf_token %}
        <!-- This is a Django template tag that includes a Cross-Site Request
Forgery (CSRF) token in the form. CSRF tokens are a security measure to protect
against malicious attacks. -->

        {{ form.as_p }}
        <!-- This is a Django template variable that renders the form fields in a
paragraph format. The 'form' variable should be defined in the associated Django
view, and it represents a form instance. The 'as_p' method renders the form
fields in a simple, paragraph-based structure. -->

        <input type="submit" value="Save" />
        <!-- This is an HTML input element of type 'submit'. It creates a button
with the label "Save" that, when clicked, submits the form to the server. -->

    </form>
{% endblock content %}
```

In summary, this Django template is creating a new article form that extends the structure of another template called 'base.html'. The form includes a heading, form fields for the article, a CSRF token for security, and a "Save" button to submit the form. The actual appearance and styling of these elements are inherited from the 'base.html' template.

## {% csrf_token %}

In web applications, there's a security vulnerability known as Cross-Site Request Forgery (CSRF), where an attacker deceives a user's browser into making unintended requests. Django addresses this concern by incorporating an inherent protective measure, and the {% csrf_token %} serves as a template tag facilitating the implementation of this protection.

When a form is submitted in Django, it includes a concealed input field housing a CSRF token. This token is distinct for each user session, ensuring that the form submission originates from the same user who initially requested the page. The {% csrf_token %} template tag essentially embeds this token into the form.

To break it down:

{% csrf_token %}: This tag generates and inserts a concealed input field containing a unique CSRF token into the HTML form.It aids Django in confirming the legitimacy of the form submission, guarding against CSRF attacks.

## {{ form.as_p }}

Django offers a robust form handling system, and {{ form.as_p }} is a method for displaying a Django form in a straightforward, paragraph-based format within a template.

Here's how it works:

{{ form }}: This is a template variable representing a Django form instance, defined and passed from a Django view.

.as_p: This method is applied to the form variable, rendering the form fields in a paragraph-based structure.

It's a convenient method for swiftly presenting form fields in a basic layout, eliminating the need to manually code each field individually. If desired, the actual appearance can be further customized using CSS. Now let’s visit: http://127.0.0.1:8000/article/new/


Write a new article and save it.


Opps!!!!....What happens?

Django's error message points out a helpful detail. It's indicating that we haven't specified the destination for the user after successfully submitting the form. To address this, we can redirect the user to the detail page upon successful form submission, allowing them to view their completed post.

Following Django's recommendation, it's advisable to include a get_absolute_url method in our model. This is considered a best practice to ensure a canonical URL is set for an object. This way, even if the URL structure changes in the future, the reference to the specific object remains the same. To implement this, it's recommended to add both a get_absolute_url() and __str__() method to every model.

To begin, navigate to the models.py file. On the second line, import the reverse function, and then add a new get_absolute_url method.

***article/models.py***

```
from django.db import models
# Import the 'reverse' function from the 'django.urls' module.
from django.urls import reverse

class Article(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
    'auth.User',
    on_delete=models.CASCADE,
    )
    body = models.TextField()

    def __str__(self):
        return self.title

    # Define a method 'get_absolute_url'. This method is often used to get the
canonical URL for an object.
    # In this case, it uses the 'reverse' function to generate a URL for the
'article_detail' view,
    # and it passes the article's ID as an argument to the URL pattern.

    def get_absolute_url(self):
        return reverse('article_detail', args=[str(self.id)])
```

In summary, this code defines a Django model called 'Article' with fields for title, author, and body. It also includes a method 'get_absolute_url' that generates the URL for viewing a specific article using the 'reverse' function. This URL is based on the 'article_detail' view, and it includes the article's ID as an argument in the URL pattern. This can be useful when you want to link directly to a specific article.

**The reverse function**

Reverse is a convenient utility function provided by Django. It enables us to reference an object using its URL template name, such as article_detail in this case. This enhances the flexibility of our application and is considered good practice.

**Usage in get_absolute_url method:**

```
    def get_absolute_url(self):
        return reverse('article_detail', args=[str(self.id)])
```

In this example, the get_absolute_url method is defined for the Article model. This method is often used to provide a canonical URL for an instance of the model.

**Arguments:**

'article_detail': This is the name of the URL pattern or the view name that you want to reverse. In your Django project's urls.py, you would have a URL pattern defined with a name, and here, 'article_detail' is assumed to be one such name.

args=[str(self.id)]: This part provides any additional arguments needed for the URL pattern. In this case, it includes the id of the current Article instance. The args parameter is a list of positional arguments that will be passed to the URL pattern.

**Return Value:**

The reverse function returns a URL as a string. This URL is dynamically generated based on the provided view name and arguments, making it easier to manage URLs and reducing the chance of errors.

That means reverse allows you to avoid hardcoding URLs in your code, making your code more maintainable and adaptable to changes in your URL patterns. It's especially useful when you want to generate URLs for specific views and include dynamic elements like object IDs.

Visit: http://127.0.0.1:8000/


Click on +New Article.


Now click on the save button.


Now we can successfully create a new article from the front end itself.

### Update Form

But what if we want to update the article?
Well we have to create a mechanism for that too. First we have to update our article_detail.html file.

***templates/article_detail.html***

```
{% extends 'base.html' %}
{% block content %}
<div class="article-entry">
<h2>{{ object.title }}</h2>
<p>{{ object.body }}</p>
</div>

<!-- The following line creates a hyperlink to the 'article_edit' view, passing
the primary key (pk) of the current article -->

<a href="{% url 'article_edit' article.pk %}">+ Edit Article </a>
```

```
{% endblock content %}
```

Now let’s create the article_edit.html file.

***command prompt***

```
>echo> templates/article_edit.html
```

Now let’s update our newly created file.

***templates/article_edit.html***

```
{% extends 'base.html' %}
{% block content %}
<h1>Edit article</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Update" />
</form>
{% endblock content %}
```

This code extends a base template, which contains the overall structure of the webpage. Inside the content block, it displays a heading "Edit article" and a form with fields for updating the article content. The form uses the POST method for submission, and a CSRF token is included for security. The form fields are rendered using the as_p method, and there's a submit button with the label "Update".

Now let’s create a view for this functionality.

***news/views.py***

```
from django.views.generic import ListView, DetailView
from .models import Article

# We're importing two classes, CreateView and UpdateView, from the
django.views.generic.edit module
from django.views.generic.edit import CreateView, UpdateView

class ArticleListView(ListView):
    model = Article
    template_name = 'home.html'


class ArticleDetailView(DetailView):
    model = Article
```

```
    template_name = 'article_detail.html'



class ArticleCreateView(CreateView):
    model = Article
    template_name = 'article_new.html'
    fields = '__all__'


# Now, let's define a new class called ArticleUpdateView. This class inherits
from the UpdateView class provided by Django.
class ArticleUpdateView(UpdateView):
    # Inside the class, we specify some attributes or properties.

    # The model attribute tells Django which model this view is going to interact
with. In this case, it's the Article model.
    model = Article

    # The template_name attribute specifies the template (HTML file) that will be
used to render this view.
    template_name = 'article_edit.html'

    # The fields attribute is a list of fields from the Article model that we
want to include in the form.
    fields = ['title', 'body']

# That's it! This class is now an UpdateView tailored to work with the Article
model.
```

In summary, this code creates and updates the views for the Article model in Django. It specifies the model, the template to use, and the fields to include in the form. The UpdateView class in Django is a generic class-based view that simplifies the creation of views for updating model instances.

Now let's update the urls.py file.

***news/urls.py***

```
from django.urls import path
# Importing necessary views from the views module

from .views import ArticleListView, ArticleDetailView, ArticleCreateView,
ArticleUpdateView
```

```
urlpatterns = [

    # Now let’s try setting up the URL pattern for editing an article
    # The 'article/<int:pk>/edit/' part defines the URL pattern
    # '<int:pk>' captures an integer value from the URL and assigns it to the
'pk' variable
    # 'ArticleUpdateView.as_view()' associates the URL pattern with the
ArticleUpdateView class
    # 'name='article_edit'' gives a name to this URL pattern for easy reference
in the code

    path('article/<int:pk>/edit/',ArticleUpdateView.as_view(),
name='article_edit'),

    path('article/new/', ArticleCreateView.as_view(), name='article_new'),

    path('article/<int:pk>/', ArticleDetailView.as_view(),
name='article_detail'),

    path('', ArticleListView.as_view(), name='home'),
]
```

Now let’s see if we can edit our existing articles.

Visit: http://127.0.0.1:8000/


Now click on any article that you want to edit.


Click on **+ Edit Article** button.


Now edit the content of the article and click on update.


Now you can see the updated article.

## Delete View

Now, if we want to delete an article, we have to create a mechanism for that too.
For that let's create a html file called article_delete.html

***command prompt***

```
>echo> templates/article_delete.html
```

Now, let's edit the article_detail.html file.

***templates/article_detail.html***

```
{% extends 'base.html' %}
{% block content %}

<div class="article-entry">
<h2>{{ object.title }}</h2>
<p>{{ object.body }}</p>
</div>

<a href="{% url 'article_edit' article.pk %}">+ Edit Article Article</a>
<p><a href="{% url 'article_delete' article.pk %}">+ Delete Article
Article</a></p>

<!-- Inside this block, there's a paragraph (<p>) containing a link (<a>). The
href attribute of the link is set using Django template syntax. {% url
'article_delete' article.pk %} generates the URL for the 'article_delete' view,
passing the primary key (pk) of thecurrent article as a parameter. This link
leads to a page where the usercan delete the article.-->

{% endblock content %}
```

Now let's update the article_delete.html file.

***templates/article_delete.html***

```
{% extends 'base.html' %}
{% block content %}

<h1>Delete article</h1>

<form action="" method="post">
```

```
<!-- This is an HTML form. The 'action' attribute is empty, which means the form
will submit to the same URL it's on. The 'method' is set to 'post', indicating
that the form data will be sent securely in the HTTP request body. -->

{% csrf_token %}
<!-- This is a Django template tag used to include a Cross-Site Request Forgery
(CSRF) token in the form. It's a security measure to protect against CSRF
attacks. -->

<p>Are you sure you want to delete "{{ article.title }}"?</p>
<!-- This is a paragraph (p) displaying a message with the title of the article.
The title is enclosed in double curly braces, indicating it's a Django template
variable. The actual value will be inserted when the template is rendered. -->

<input type="submit" value="Confirm" />
<!-- This is an HTML input element of type 'submit', which means it's a button
that, when clicked, will submit the form. The text on the button is "Confirm". --
>

</form>
<!-- This tag marks the end of the form. -->

{% endblock content %}
```

This HTML file extends a base template ('base.html'). It includes a form for confirming the deletion of an article. The form uses the POST method for secure data submission and includes a CSRF token for protection. The title of the article is displayed in the confirmation message, and users can confirm the deletion by clicking the "Confirm" button.

Now let's update the views.py

***news/views.py***

```
from django.views.generic import ListView, DetailView
from .models import Article

# Import specific generic views for creating, updating, and deleting object
from django.views.generic.edit import CreateView,  UpdateView, DeleteView

# Import a utility function for reversing URL patterns
from django.urls import reverse_lazy
```

```
class ArticleListView(ListView):
    model = Article
    template_name = 'home.html'


class ArticleDetailView(DetailView):
    model = Article
    template_name = 'article_detail.html'



class ArticleCreateView(CreateView):
    model = Article
    template_name = 'article_new.html'
    fields = '__all__'


class ArticleUpdateView(UpdateView):
    model = Article
    template_name = 'article_edit.html'
    fields = ['title', 'body']


# let’s define a class for a view that handles deleting Article objects
class ArticleDeleteView(DeleteView):
    # let’s specify the model that this view is working with (Article in this
case)
    model = Article

    # let’s specify the HTML template that will be used to render this view
    template_name = 'article_delete.html'

    # let’s specify the URL to redirect to upon successful deletion (using
reverse_lazy for delayed evaluation)
    success_url = reverse_lazy('home')
```

We have introduced a new term here called reverse_lazy. Let’s talk about it.

**reverse_lazy**

reverse_lazy in Django is a function used for delayed URL resolution. When defining success or cancel URLs in views, it ensures that the URL is not evaluated until the view is triggered. This is especially useful for class-based views where URLs are not available at the time of importing. By delaying the URL resolution until it's actually needed, it helps prevent circular import issues and ensures that the correct

URL is used when the view is executed, providing a cleaner and more reliable way to handle URL redirections in your Django application.

Now let's update the urls.py file.

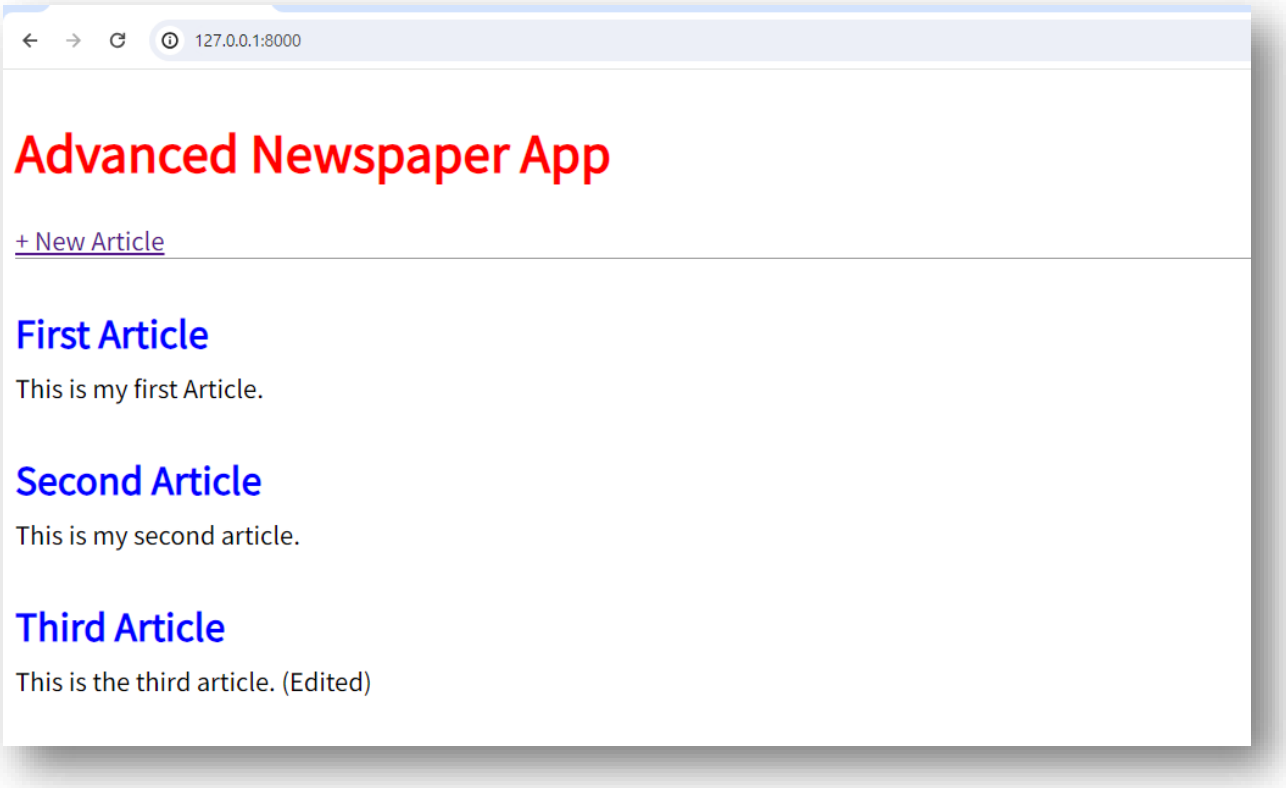
***news/urls.py***

```
from django.urls import path

# Importing necessary views from the views file in the same directory
from .views import ArticleListView, ArticleDetailView, ArticleCreateView,
ArticleUpdateView, ArticleDeleteView

urlpatterns = [

    # The 'article/<int:pk>/delete/' part defines the URL pattern

    # '<int:pk>' is a path converter that matches an integer and passes it as a
variable named 'pk' to the view

    # 'ArticleDeleteView.as_view()' associates this URL pattern with the
ArticleDeleteView class

    # 'name='article_delete'' provides a unique identifier for this URL pattern

    path('article/<int:pk>/delete/',ArticleDeleteView.as_view(),
name='article_delete'),

    path('article/<int:pk>/edit/',ArticleUpdateView.as_view(),
name='article_edit'),

    path('article/new/', ArticleCreateView.as_view(), name='article_new'),

    path('article/<int:pk>/', ArticleDetailView.as_view(),
name='article_detail'),

    path('', ArticleListView.as_view(), name='home'),
]
```

So, in simple terms, this is responsible for defining a URL pattern for deleting an article. The URL pattern includes a placeholder for the article's primary key (<int:pk>), and when a user accesses a URL matching this pattern, it will trigger the ArticleDeleteView class to handle the deletion process. The name='article_delete' is a unique identifier for this URL pattern, which can be used to reference it in other parts of the code, such as templates or other views.

Now let’s check if we can delete an article.

Visit: http://127.0.0.1:8000/




Click on any aritcle.


Click on **+ Delete Article** button.


Now click on **Confirm** button.


Now you can see that the article is deleted.

### Tests

Now let's create tests for our updated application.

> Occasionally, you may observe incorrect Python code indentation due to width constraints on this page. To rectify this issue, kindly copy the code and paste it into a text editor such as VS Code. Doing so should result in the correct indentation.

**news/tests.py**

```
from django.contrib.auth import get_user_model
from django.test import Client, TestCase
from django.urls import reverse
from .models import Article

class ArticleTests(TestCase):

    def setUp(self):
        self.user = get_user_model().objects.create_user(
            username='dreamer',
```

```
            email='dreamer@email.com',
            password='secret'
        )
        self.article = Article.objects.create(
            title='Enchanting Adventures in Wonderland',
            body='Embark on a magical journey through the whimsical Wonderland,
where talking rabbits and mad hatters await!',
            author=self.user,
        )

    def test_string_representation(self):
        article = Article(title='Magical Mysteries Unveiled')
        self.assertEqual(str(article), article.title)

    def test_article_content(self):
        self.assertEqual(f'{self.article.title}', 'Enchanting Adventures in
Wonderland')
        self.assertEqual(f'{self.article.author}', 'dreamer')
        self.assertEqual(f'{self.article.body}', 'Embark on a magical journey
through the whimsical Wonderland, where talking rabbits and mad hatters await!')

    def test_article_list_view(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Embark on a magical journey through the
whimsical Wonderland')
        self.assertTemplateUsed(response, 'home.html')

    def test_article_detail_view(self):
        response = self.client.get(reverse('article_detail',
args=[str(self.article.id)]))
        no_response = self.client.get(reverse('article_detail', args=[999]))  #
Assuming an invalid ID
        self.assertEqual(response.status_code, 200)
        self.assertEqual(no_response.status_code, 404)
        self.assertContains(response, 'Enchanting Adventures in Wonderland')
        self.assertTemplateUsed(response, 'article_detail.html')

    def test_article_create_view(self):
        response = self.client.post(reverse('article_new'), {
            'title': 'Discovering Atlantis',
            'body': 'Uncover the secrets of the lost city of Atlantis and its
mythical wonders!',
            'author': self.user.id,
        })
```

```
        self.assertEqual(response.status_code, 302)  # Redirects after successful
form submission
        self.assertTrue(Article.objects.filter(title='Discovering
Atlantis').exists())

    def test_article_update_view(self):
        response = self.client.post(reverse('article_edit',
args=[str(self.article.id)]), {
            'title': 'Journey to the Stars',
            'body': 'Embark on a cosmic adventure as you explore the mysteries of
distant galaxies!',
        })
        self.assertEqual(response.status_code, 302)  # Redirects after successful
form submission
        self.article.refresh_from_db()
        self.assertEqual(self.article.title, 'Journey to the Stars')
        self.assertEqual(self.article.body, 'Embark on a cosmic adventure as you
explore the mysteries of distant galaxies!')

    def test_article_delete_view(self):
        response = self.client.post(reverse('article_delete',
args=[str(self.article.id)]))
        self.assertEqual(response.status_code, 302)  # Redirects after successful
deletion
        self.assertFalse(Article.objects.filter(title='Enchanting Adventures in
Wonderland').exists())
```

This tests.py file includes test cases for string representation, content, list view, detail view, create view, update view, and delete view for the Article model and views in your Django app.

To execute the tests, stop the server using Control+c, and enter the command "python manage.py test" in the command line:

***command prompt***

```
> python manage.py test
```

You will get a similar output as following:
Output:

***command prompt***

```
Found 7 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.......
```

```
----------------------------------------------------------------------
Ran 7 tests in 6.717s

OK
Destroying test database for alias 'default'...
```

## Conclusion

This chapter extensively covered key aspects of web development using Django, with a specific focus on Forms and User Input, along with a thorough examination of the foundational principles underlying CRUD operations. The ability to adeptly gather, validate, and process user input through forms is emphasized as a critical skill for constructing dynamic and engaging web applications. Furthermore, we delved into the central role of CRUD functionalities in database operations, underscoring their importance in ensuring effective data persistence.

Looking ahead in our Django journey, the insights acquired in this chapter provide a sturdy groundwork. Empowered with the skills to seamlessly handle user input and manipulate database data, we are well-prepared to craft robust and user-centric web applications. This knowledge positions us for success as we progress further into the intricacies of Django development.

# Chapter 7: User Account

Currently, we only have a single user, the Super user, generated through the command prompt. Now, our goal is to implement functionality that enables regular users to create their accounts directly from the front end of our website. To begin, let's initiate the process by creating a project.

Go to the *django_projects* Folder. Create a folder *user_registration*.

***command prompt***

```
> cd Desktop/django_projects
> mkdir user_registration
> cd user_registration
```

Now, Inside our *user_registration* folder Create a virtual environment and install Django in it.

***command prompt***

```
> pipenv shell
> pip install django
```

Let’s create the Django project.

***command prompt***

```
> django-admin startproject user_registration .
```

Now let’s start an app and migrate our apps.

***command prompt***

```
> python manage.py startapp userR
> python manage.py migrate
```

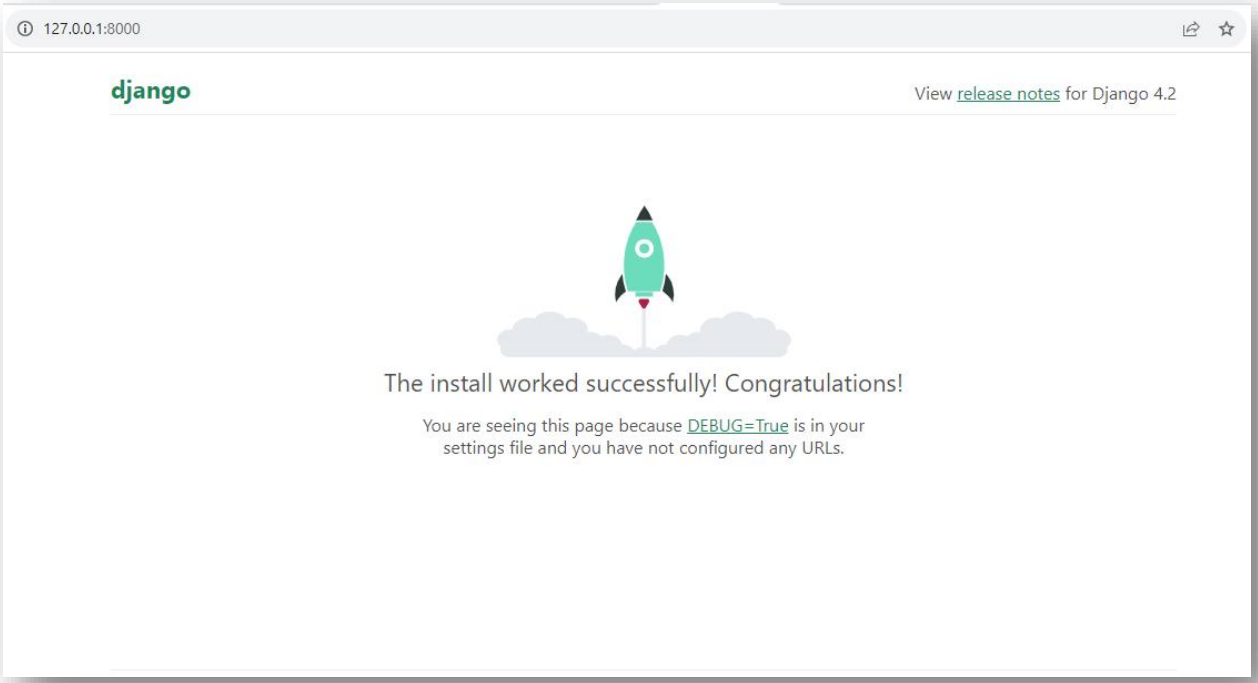
After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

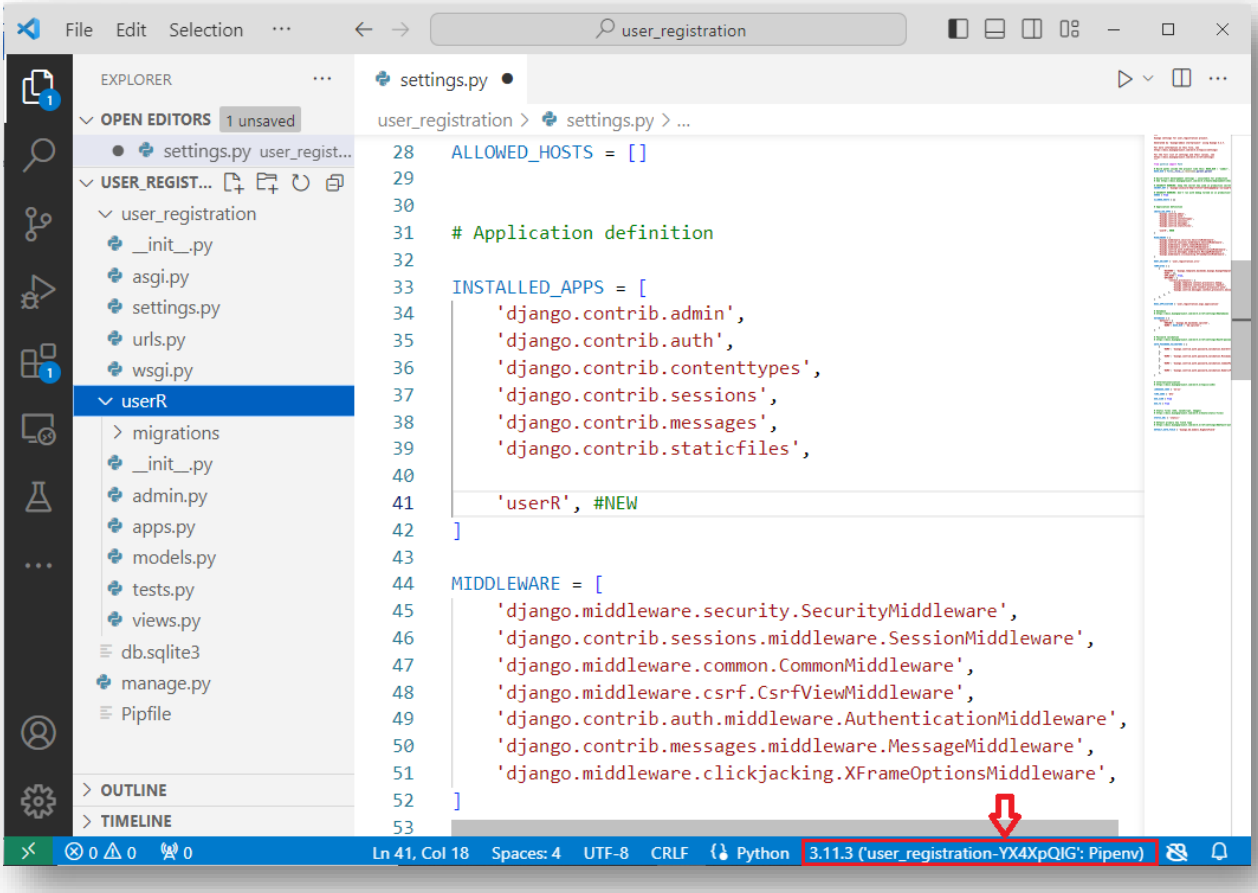
```
> python manage.py runserver
```

Now Visit: http://127.0.0.1:8000/

You will able to see:




Now open the project in vs code and confirm that the correct interpreter is selected.




Now Let's work on the setting.py and add the newly created app there.

***settings.py***

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'userR',
]
```

Now let's create a super user.

***command prompt***

```
> python manage.py createsuperuser
> Username (leave blank to use 'pc'): su1
```

```
> Email address: su1@email.com
> Password:
> Password (again):
> Superuser created successfully.
```

Select any username, email, and password of your choice. Ensure to remember the password you've chosen. For instance, my password is "testuser@111".

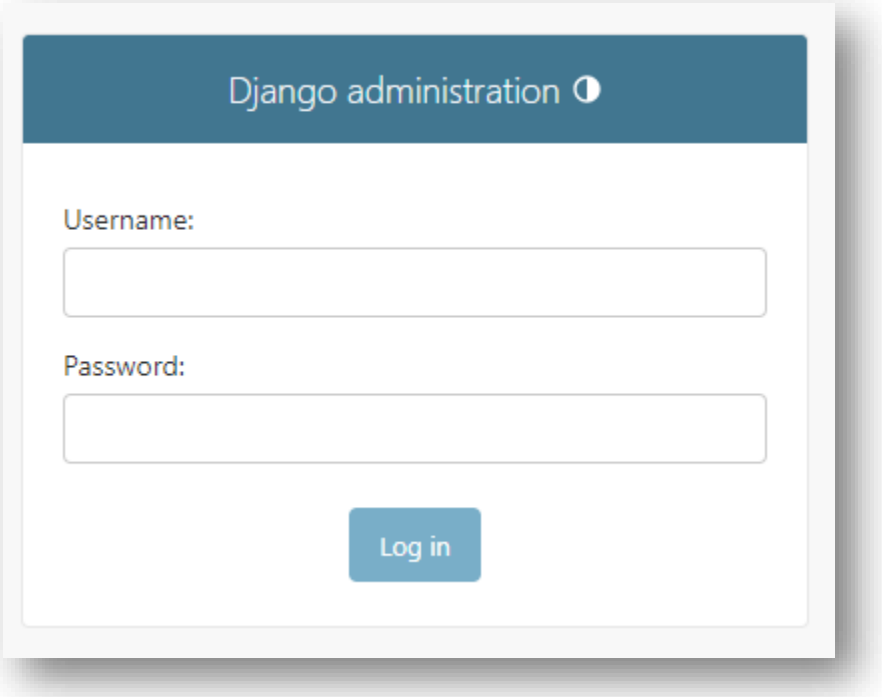
Once these tasks are done, if the server is currently active, halt it by using Control+C. Subsequently, input the command "python manage.py runserver" in the command line:

***command prompt***

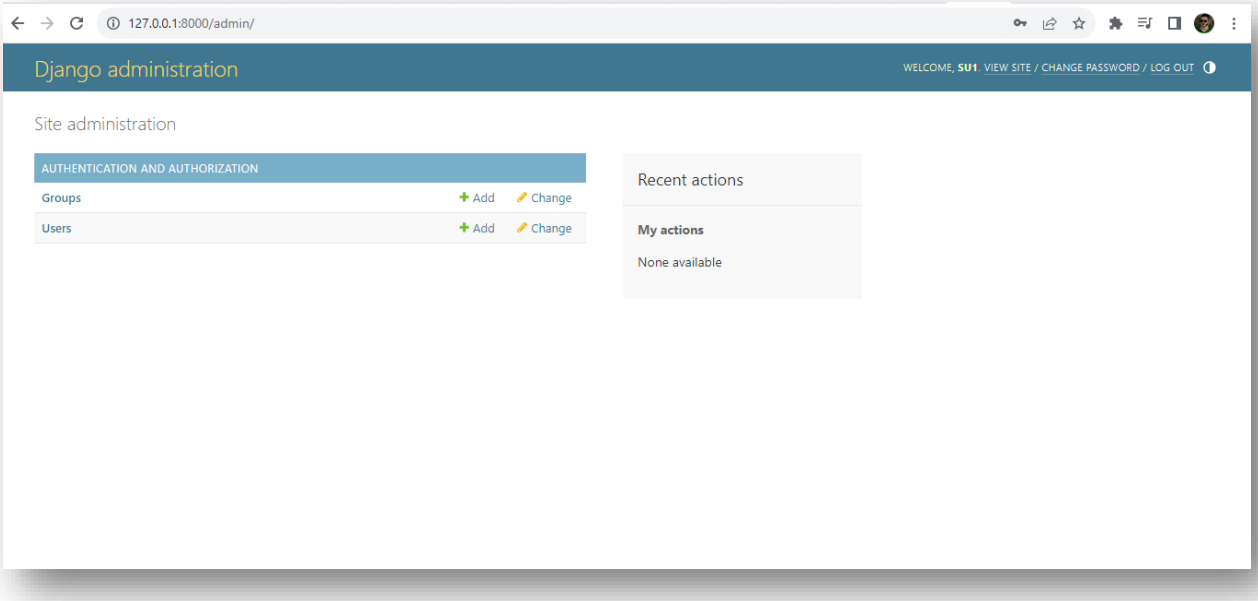
```
> python manage.py runserver
```

To log in as a superuser, visit: http://127.0.0.1:8000/admin/

You will see a login form, please enter the username and password you have just created here.




You will see something like this there.




Now, Let's build a structure for the user registration.

***command prompt***

```
> echo> userR/urls.py
```

let's update the urls.py file

***userR/urls.py***

```
# Here, we're  importing 'TemplateView' from 'django.views.generic'.
from django.urls import path
from django.views.generic import TemplateView

urlpatterns = [
    # This line is defining a URL pattern using the 'path' function.
    # The first argument ('') represents the URL path.
    # In this case, it's an empty string, which means it's the root URL.
    # The second argument is using 'TemplateView.as_view()' to render a template.
    # 'template_name' specifies the name of the template file, in this case,
'home.html'.
    # Finally, 'name' is a unique identifier for this URL pattern.
    path('', TemplateView.as_view(template_name='home.html'), name='home'),
]
```

***user_registration/urls.py***

```
# This line imports the admin module from django.contrib, which allows us to set
up the Django admin interface.
from django.contrib import admin

# This line imports the path and include modules from django.urls, which are used
to define URL patterns.
from django.urls import path, include

urlpatterns = [
    # This line maps the URL '/admin/' to the admin.site.urls, so when you go to
'/admin/', you access the Django admin interface.
    path('admin/', admin.site.urls),

    # This line includes the URLs provided by django.contrib.auth.urls.
    # It's a convenient way to include authentication-related URLs such as login,
logout, and password reset.
    # The 'userR/' prefix is added to distinguish these authentication URLs from
other URLs in your project.
    path('userR/', include('django.contrib.auth.urls')),

]
```

Now let's create the template files.

***command prompt***

```
> mkdir templates
> echo>  templates/base.html
> echo>  templates/home.html
> mkdir templates\registration
> echo> templates\registration\login.html
```

Now let's update the base.html and home.html

***templates/base.html***

```
<html>
<head>
<title>User Registration</title>
</head>
<body>
<header>
<h1><a href="{% url 'home' %}">Home Page</a></h1>
</header>
<div>
```

```
{% if user.is_authenticated %}
    <!-- This checks if the user is authenticated, meaning they are logged in. --
>
    <p>Hello {{ user.username }}!</p>
    <!-- If authenticated, it displays a greeting with the username of the
logged-in user. -->
    <p><a href="{% url 'logout' %}">Log out</a></p>
    <!-- It also provides a link to the 'logout' URL, allowing the user to log
out. -->
{% else %}
    <!-- If the user is not authenticated (not logged in), it shows this block. -
->
    <p>Please login.</p>
    <!-- Displays a message prompting the user to log in. -->
    <a href="{% url 'login' %}">Log In</a>
    <!-- Provides a link to the 'login' URL for the user to log in. -->
{% endif %}
<!-- The endif marks the end of the if-else block in Django templates. -->
{% block content %}
{% endblock content %}
</div>
</body>
</html>
```

Now let's update he home.html

***templates/home.html***

```
{% extends 'base.html' %}
{% block content %}

<p>Welcome to our website. Your adventure starts now.</p>

{% endblock content %}
```

We know that in Django, the base template serves as the foundation, providing the overall structure and common elements, while child templates act as customizable modules that define the unique content of each page.

***templates/registration/login.html***

```
{% extends 'base.html' %}
{% block content %}
<h2>Log In</h2>
<form method="post">
```

```
{% csrf_token %}
{{ form.as_p }}
<button type="submit">Log In</button>
</form>
{% endblock content %}
```

The provided code excerpt represents a Django template responsible for rendering a login form. It establishes a content block where the login form will be integrated.

It includes security measures like the CSRF token and allows users to input data, which can be sent to the server for user authentication.

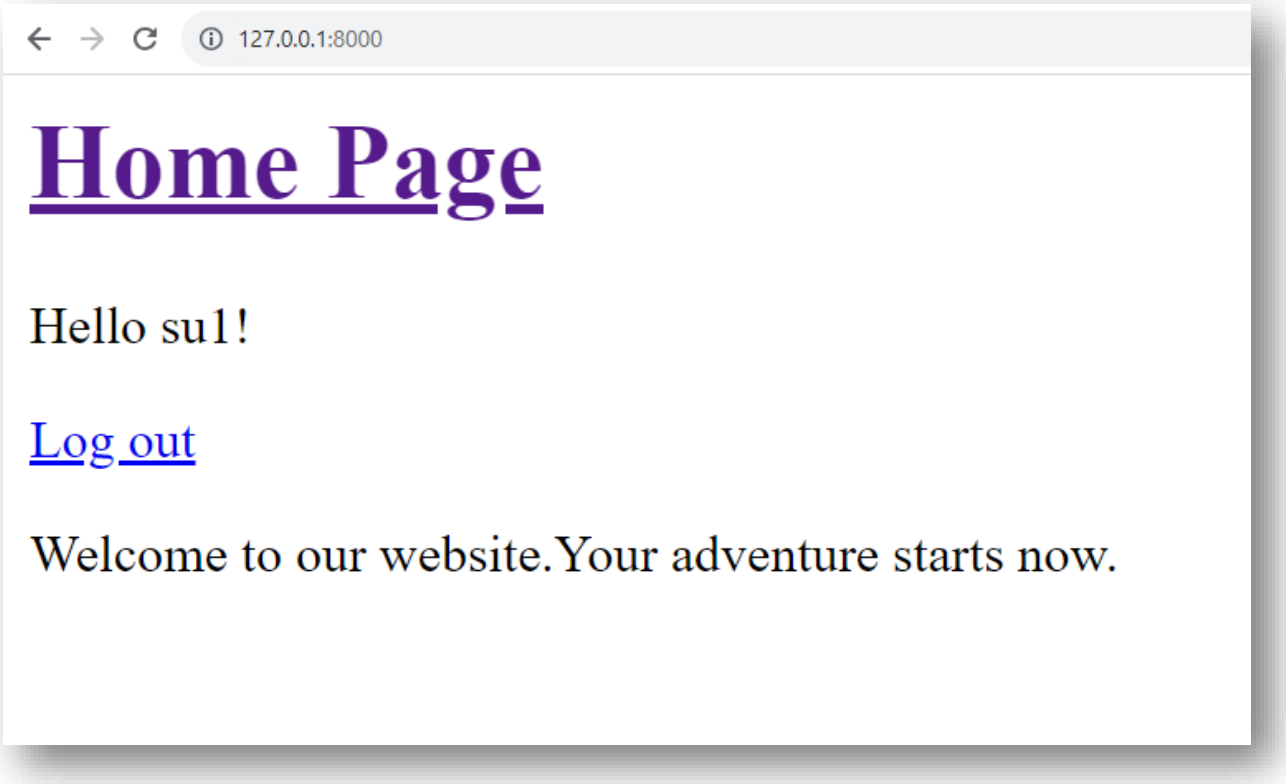
Now let's update the settings.py

***settings.py***

```
from pathlib import Path
import os


TEMPLATES = [
    {
        …
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        …
    },
]

# LOGIN_REDIRECT_URL is another setting in Django.
# When a user logs in, Django will redirect them to the URL specified here.
# In this case, it's set to 'home', which suggests that after logging in, the
user should be redirected to the 'home' page.

LOGIN_REDIRECT_URL = 'home'
# This line sets the URL where the user will be redirected after a successful
logout.
# Similarly, after logging out, the user will be redirected to the 'home' URL.
LOGOUT_REDIRECT_URL = 'home'
```

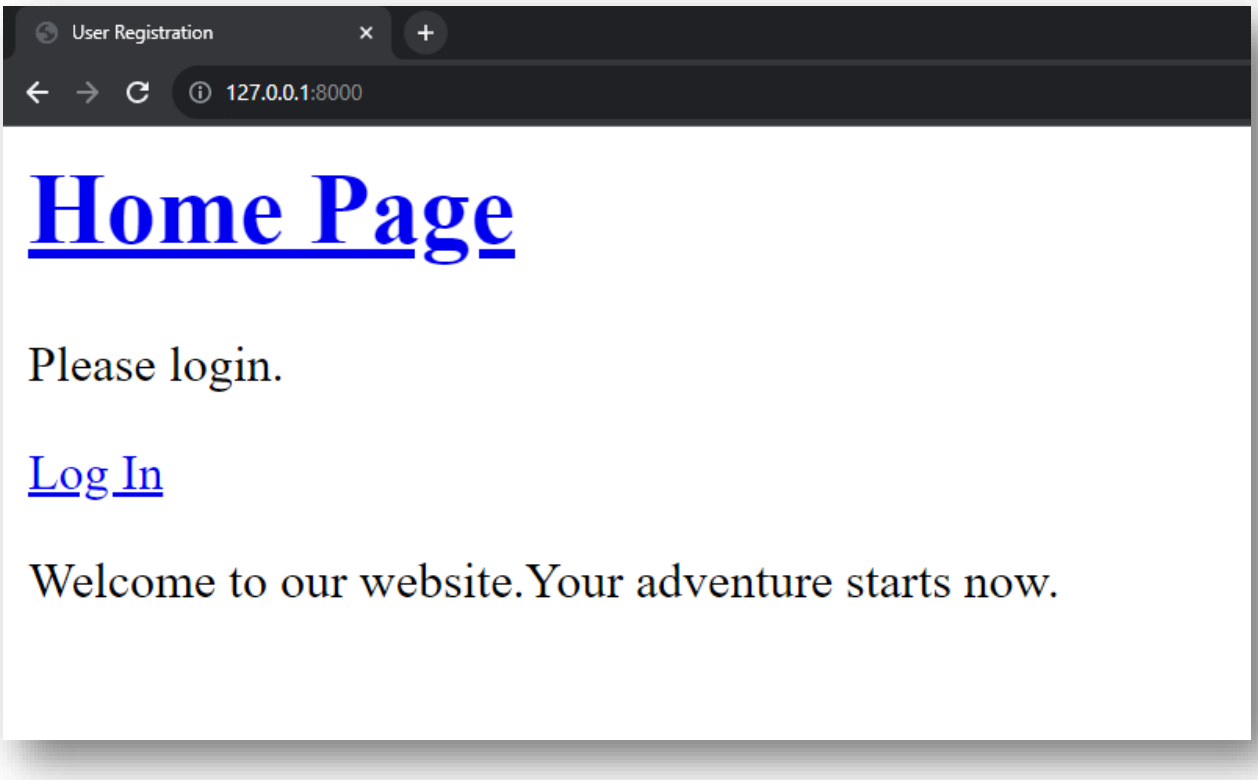
Now Visit: http://127.0.0.1:8000/

Do you see the Home page there?

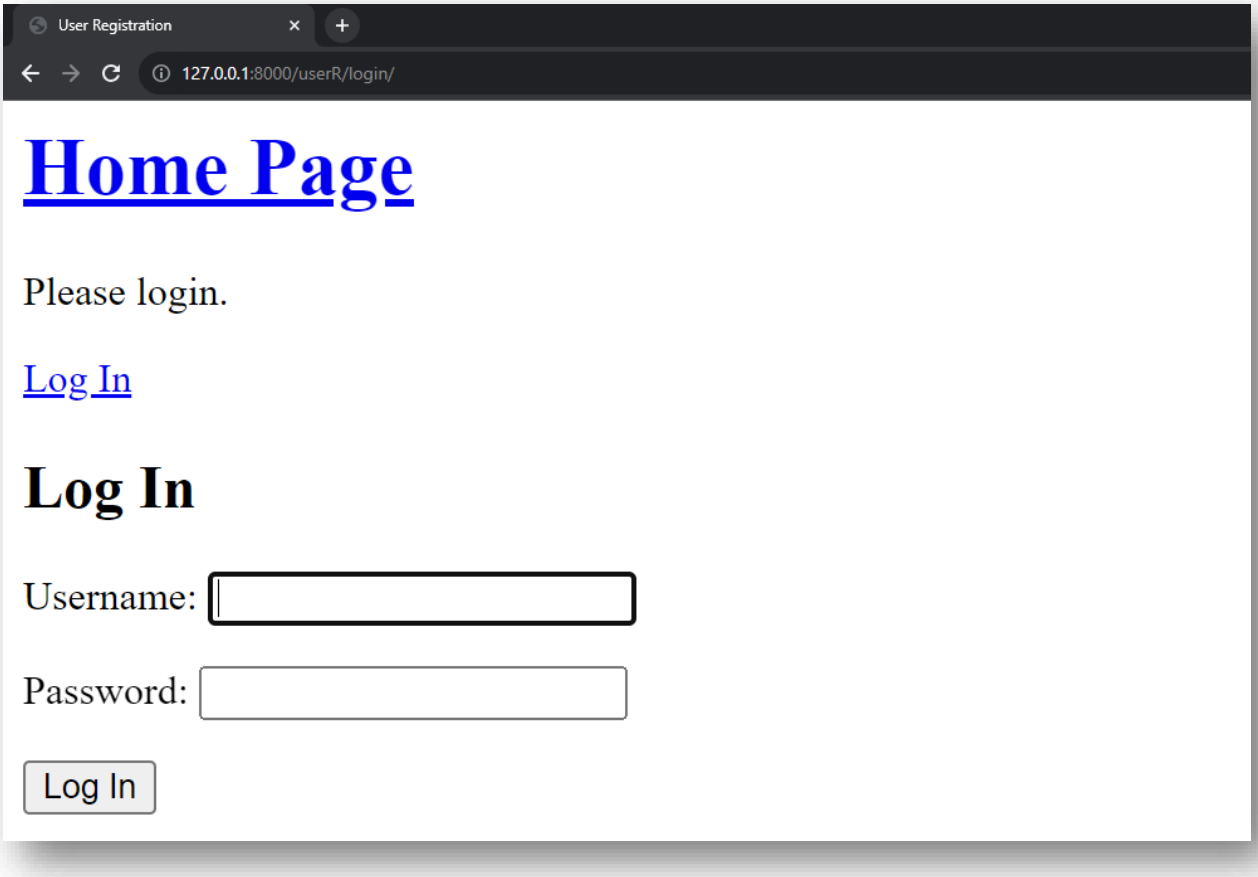



Here, you already log in as a super user. So to see the logged-out page, open the link in the incognito mode.

Link: http://127.0.0.1:8000/




Now, click on the log-in and enter your super user details.




You have successfully logged in, but please note that the signup option is currently unavailable. Let's proceed to create the signup form.

***userR/views.py***

```
# We're importing the UserCreationForm from the django.contrib.auth.forms module.
from django.contrib.auth.forms import UserCreationForm

# We're importing the reverse_lazy function from the django.urls module.
from django.urls import reverse_lazy

# We're importing the generic views from the django.views module.
from django.views import generic

# We're defining a new class called SignUpView, which is a generic CreateView.
class SignUpView(generic.CreateView):
    # Here, we're specifying the form class to be used for user registration,
    # and it's the built-in UserCreationForm provided by Django.
    form_class = UserCreationForm
```

```
    # We're setting the success URL, which is where the user will be redirected
    # after successfully signing up. In this case, it's the 'login' URL.
    success_url = reverse_lazy('login')

    # This is the name of the template file that will be used to render the
signup page.
    template_name = 'signup.html'
```

Now let's create the signup.html file.

***command prompt***

```
> echo>  templates/signup.html
```

Now let's update the file.

***templates/signup.html***

```
{% extends 'base.html' %}
{% block content %}
<h2>Sign up</h2>
<form method="post">
{% csrf_token %}
{{ form.as_p }}
<button type="submit">Sign up</button>
</form>
{% endblock content %}
```

This Django template excerpt is designed to display a straightforward sign-up form. It extends the base.html template.

This code excerpt illustrates the foundational structure of Django templates for rendering forms. The use of the form.as_p tag streamlines the rendering of individual form fields, simplifying the creation of forms with diverse input fields and validation rules.

***templates/base.html***

```
<html>
<head>
<title>User Registration</title>
</head>
<body>
<header>
<h1><a href="{% url 'home' %}">Home Page</a></h1>
```

```
</header>
<div>

    {% if user.is_authenticated %}
    <p>Hello {{ user.username }}!</p>
    <p><a href="{% url 'logout' %}">Log out</a></p>
    {% else %}
    <p>Please login.</p>
    <a href="{% url 'login' %}">Log In</a>

<!-- The following line creates a paragraph (<p>) with a link (<a>) to the
'signup' URL. -->

    <p><a href="{% url 'signup' %}">Sign UP</a></p>
    {% endif %}

{% block content %}
{% endblock content %}
</div>
</body>
</html>
```

In the case of non-authenticated users, a message prompts them to either log in or sign up.

***command prompt***

```
> echo> userR/urls.py
```

Now let's update the urls.py file

***userR/urls.py***

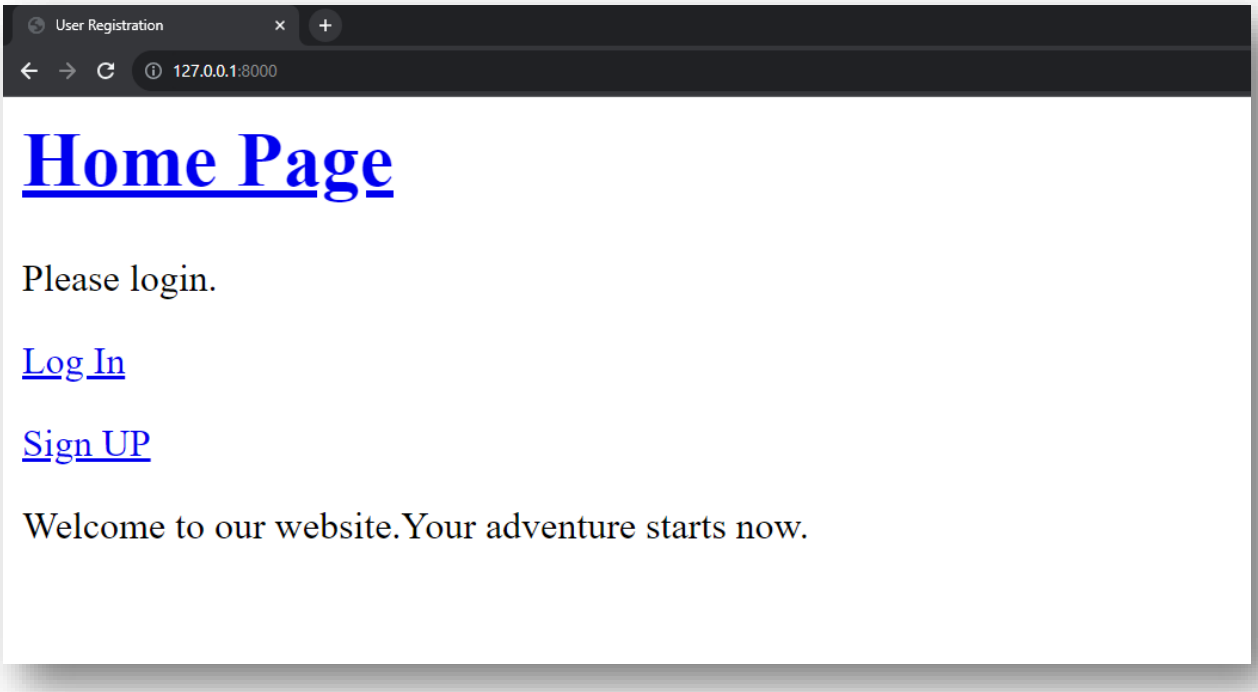
```
from django.urls import path
from django.views.generic import TemplateView
from .views import SignUpView # Importing SignUpView from the views module in the
current directory (denoted by '.')


urlpatterns = [

path('', TemplateView.as_view(template_name='home.html'), name='home'),

    # Define a URL pattern for the signup page ('signup/') using SignUpView
    # When someone visits the '/signup/' URL, Django will render the SignUpView
    # The name 'signup' can be used to refer to this URL pattern in the Django
application
```

```
path('signup/', SignUpView.as_view(), name='signup'),

]
```

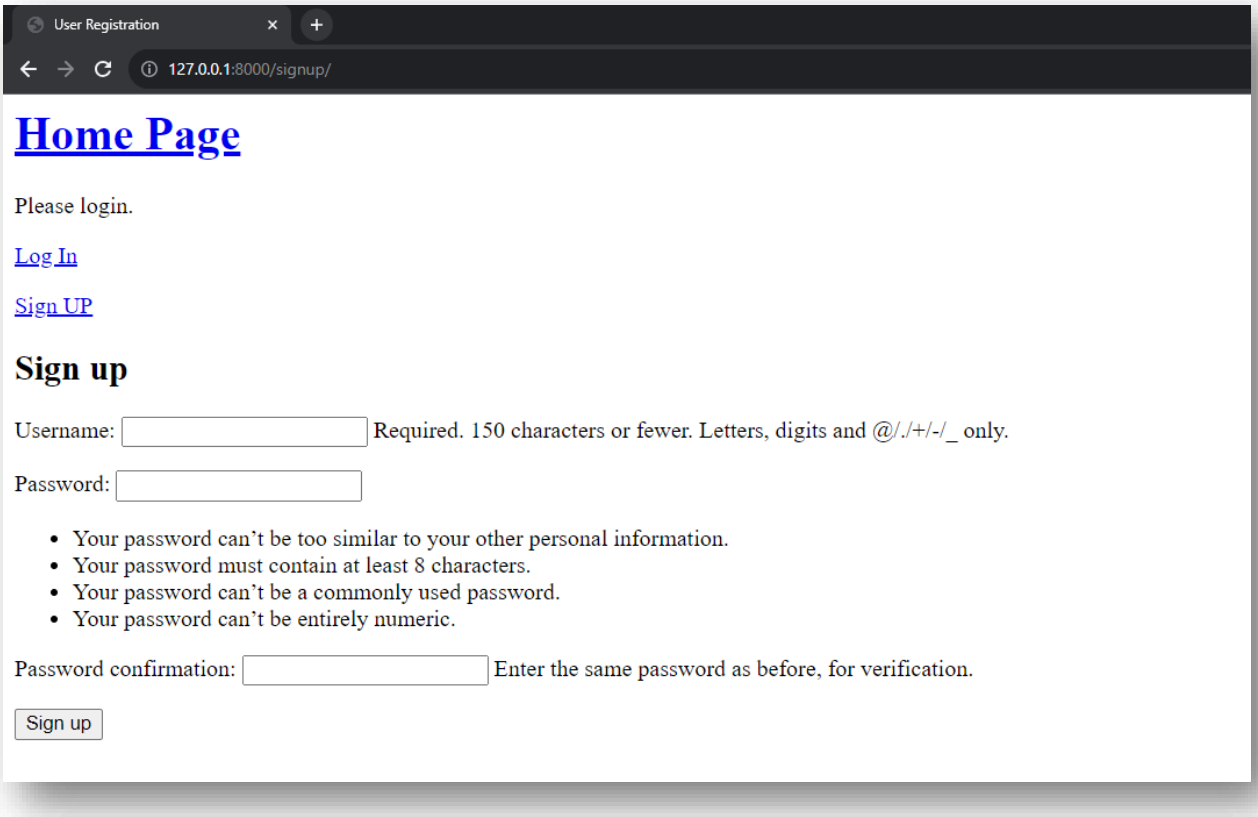
***user_registration/urls.py***

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('userR/', include('django.contrib.auth.urls')),

# An empty path ('') is the base URL. Here, we include URLs from 'userR.urls'.
# This means that if the user visits the base URL, the application will look for
URLs in 'userR.urls'.
# The 'userR.urls' file is expected to have further URL configurations for our
application.

    path('', include('userR.urls')),
]
```

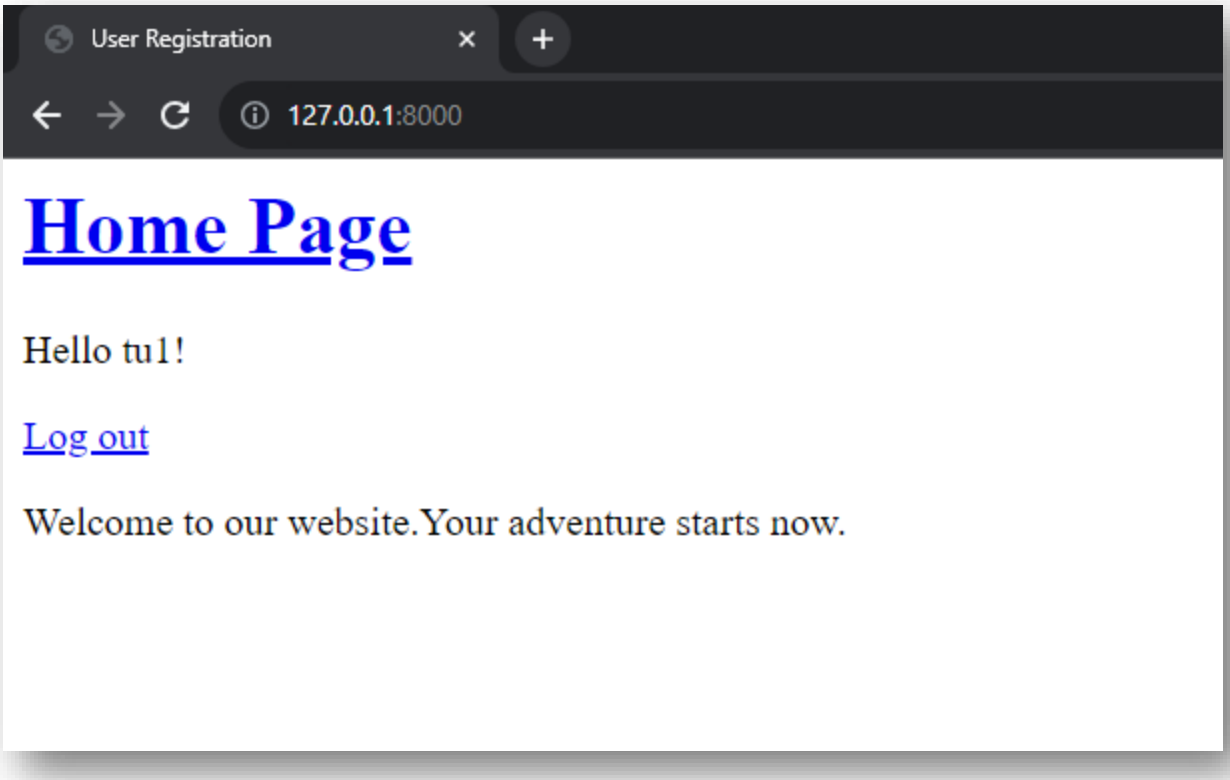
Now visit: Link: http://127.0.0.1:8000/



Click on signup and fill the form.




Now login with your new credentials.




Now you are logged in and you must be seeing your user name. That is how you create the user Registration on your django website.

## Conclusion

In conclusion, this chapter explored key aspects of user accounts in Django, covering user model creation, authentication, and security measures. We gained insights into building robust user systems, managing sessions, and implementing best practices. Armed with this knowledge, we are now equipped to design secure and user-friendly web applications using Django, setting the stage for advanced feature development in subsequent chapters.

# Chapter 8: Custom User Model

Currently, our web application enables users to finalize the registration process, but its functionality is restricted to registering users using the default Django application model.

Creating a personalized user model that better suits our specific needs is an option. Some may question the need for a custom user model when the Django default model is at hand. Nevertheless, there are compelling reasons to choose a custom user model.

Although Django's built-in models are robust and handy for numerous applications, there are situations where opting for a custom user model makes more sense than relying on the default User model provided by Django.

Here are some reasons for using a custom user model:

**Additional Fields:** If we need to store additional information about our users that is not provided by the default User model (e.g., a user profile picture, additional contact information), creating a custom user model allows us to define these extra fields in our model.

**Flexibility:** A custom user model gives us more flexibility and control over the authentication process. We can customize the authentication method, add additional methods, or override existing ones to suit the specific requirements of our application.

**Scalability:** If we anticipate the need for scalability and plan to use a different authentication method (e.g., email-based authentication instead of the default username/password), a custom user model allows us to implement these changes more seamlessly.

**Third-party Integrations:** If we are integrating with third-party authentication providers or systems, a custom user model can be adapted to work seamlessly with those systems.

**Consistency Across Apps:** If our project involves multiple Django apps that share a common user model, using a custom user model can help maintain consistency and avoid duplication of user-related functionality.

**Future-Proofing:** Designing a custom user model from the beginning can help future-proof our application. If we start with the default User model and later realize we need to customize it, migrating to a custom user model can be challenging.

***Note:*** *When initiating new projects, the official Django documentation strongly advises the utilization of a custom user model.*

No more delays.

It's time to commence the development of our tailored user model. In Django, the customary approach involves inheriting from either the **AbstractUser** or **AbstractBaseUser** classes within the auth module. Following that, we define our personalized user model in the Django project settings.

However, before delving into that, let's embark on the creation of a fresh project and app. For this instance, we'll establish a job portal website named JobVista.

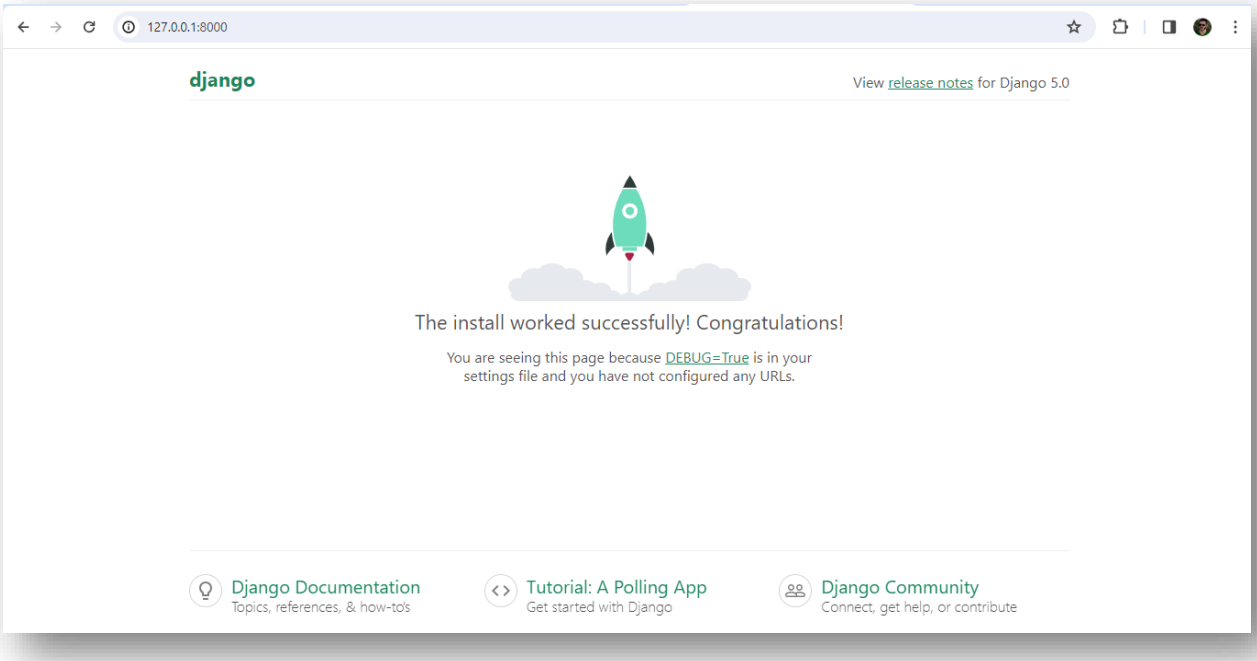
***command prompt***

```
> cd Desktop/django_pojects
> mkdir JobVista
> cd JobVista
> pipenv install django
> pipenv shell
> django-admin startproject JobVista .
> python manage.py startapp users
> python manage.py runserver
```

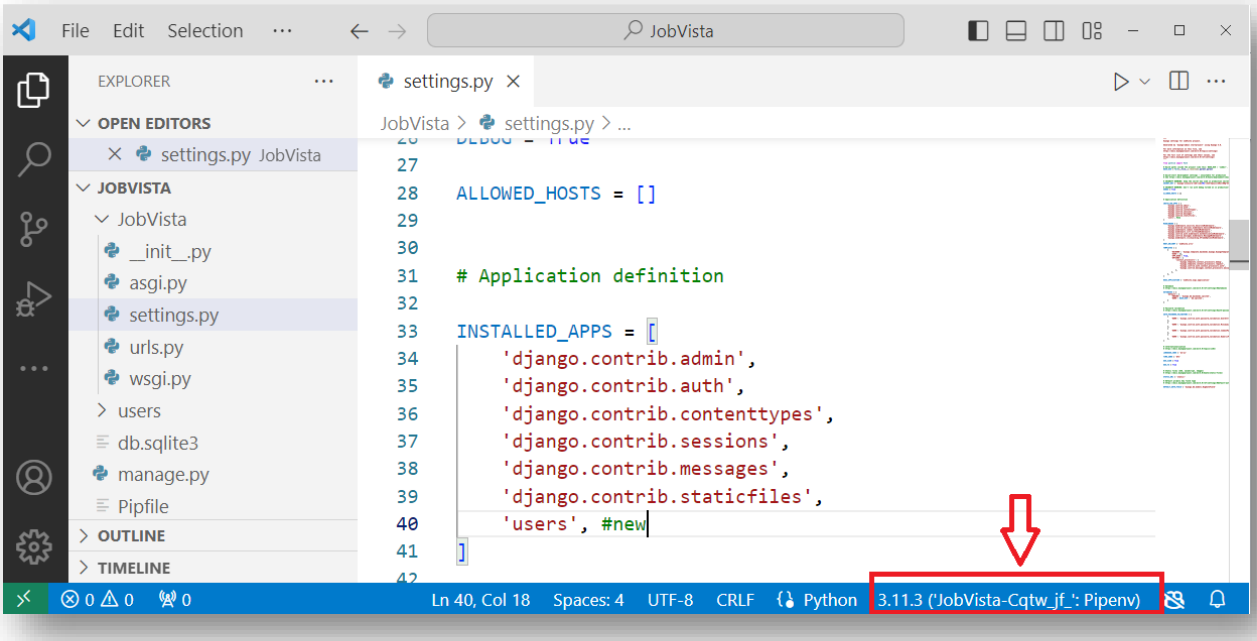
Kindly note that the migration process for configuring our database has not been initiated yet. It is essential to postpone this task until the new custom user model is created, given its tight integration with the overall Django structure.

Now visit: http://127.0.0.1:8000/

You'll encounter the Django welcome screen.




To leave the server, press: Ctrl + C. Next, launch the project in VS Code and verify that the corresponding Python virtual environment is active.



Let's proceed to modify the settings.py file by incorporating the newly developed app and specifying AUTH_USER_MODEL. Firstly, add the users app to the INSTALLED_APPS section in settings.py.

Subsequently, at the end of the file, employ the AUTH_USER_MODEL configuration to instruct Django to use our custom user model, named CustomUser, located within the users app, which is referenced as users.CustomUser.

***settings.py***

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'users',
]

AUTH_USER_MODEL = 'users.CustomUser'
```

Next, let's make modifications to the models.py file within the users app. This is where we'll manage the information of companies interested in posting job listings on the portal.

Our initial step involves introducing a new field named founded_in, dedicated to storing the year of the company's establishment.

***users/models.py***

```
# We're importing necessary components from Django to create a custom user model.
from django.contrib.auth.models import AbstractUser
from django.db import models

# Here, we are creating a new class called CustomUser that extends Django's
built-in AbstractUser class.
class CustomUser(AbstractUser):
    # We're adding a new field called 'founded_in' to our custom user model.
    # This field is of type PositiveIntegerField, which means it can only store
positive integers.
    # 'null=True' allows the field to be empty in the database (it can have a
null value).
    # 'blank=True' means that the field is not required in forms (it can be left
empty by the user).
    founded_in = models.PositiveIntegerField(null=True, blank=True)
```

Django comes with a default User model that includes fields like username, email, password, etc. However, sometimes we might need additional information about our users. This code demonstrates how to create a custom user model by extending Django's built-in AbstractUser class.

By creating a custom user model, we have the flexibility to include additional fields that are relevant to your application's requirements. This is a common practice in Django projects to tailor the user model to suit specific needs. Now let's create the forms.py to store the additional fields.

***command prompt***

```
> echo> users/forms.py
```

Now let's update forms.py. As we integrate a new field into an existing default Django form, it is necessary to define and configure it in both the user creation form and the user change form.

***users/forms.py***

```
# Import necessary modules and classes from Django
from django import forms
from django.contrib.auth.forms import UserCreationForm, UserChangeForm

# Import the CustomUser model from the current application's models
from .models import CustomUser

# Define a new form class for user creation, extending Django's built-in
UserCreationForm
class CustomUserCreationForm(UserCreationForm):

    # Define a nested Meta class to provide additional information about the form
    class Meta(UserCreationForm.Meta):
        # Set the model to be used by the form to our CustomUser model
        model = CustomUser
        # Specify the fields to be included in the form, adding 'founded_in' to
the existing fields
        fields = UserCreationForm.Meta.fields + ('founded_in',)

# Define a new form class for user modification, extending Django's built-in
UserChangeForm
class CustomUserChangeForm(UserChangeForm):

    # Define a nested Meta class to provide additional information about the form
    class Meta:
        # Set the model to be used by the form to our CustomUser model
        model = CustomUser
```

```
        # Specify the fields to be included in the form, using the fields from
UserChangeForm.Meta
        fields = UserChangeForm.Meta.fields
```

In summary, these forms are used to facilitate user registration (CustomUserCreationForm) and user profile modification (CustomUserChangeForm). They extend Django's built-in forms, adding or modifying fields as needed for the custom user model. The custom field founded_in is included in the registration form but not in the profile modification form.

*The Meta class in Django is a special class that allows you to provide additional information about a model or a form. It's not required, but it's a convenient way to include metadata or configuration options for the model or form.*

Now, let's update the users/admin.py

***users/admin.py***

```
# Import necessary modules from Django
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin

# Import custom forms and model from the same Django app (assuming they are in
the same directory)
from .forms import CustomUserCreationForm, CustomUserChangeForm
from .models import CustomUser

# Create a custom admin class for our CustomUser model
class CustomUserAdmin(UserAdmin):
    # Specify the form to be used for adding new users
    add_form = CustomUserCreationForm
    # Specify the form to be used for editing existing users
    form = CustomUserChangeForm
    # Specify the model that this admin class is associated with
    model = CustomUser

# Register the CustomUser model with the CustomUserAdmin class in the Django
admin site
admin.site.register(CustomUser, CustomUserAdmin)
```

In summary, this code is customizing the Django admin interface for the CustomUser model by using custom forms (CustomUserCreationForm and CustomUserChangeForm) and a custom admin class (CustomUserAdmin).

Now, let’s make the migrations.

***command prompt***

```
> python manage.py makemigrations users
> python manage.py migrate
```

Now, let’s create a superuser.

***command prompt***

```
> python manage.py createsuperuser
> Username (leave blank to use 'pc'): su1
> Email address: su1@email.com
> Password:
> Password (again):
> Superuser created successfully.
```

You can choose any username, email, or password.

***Note:*** *Passwords won’t be visible on the screen for security reasons.*

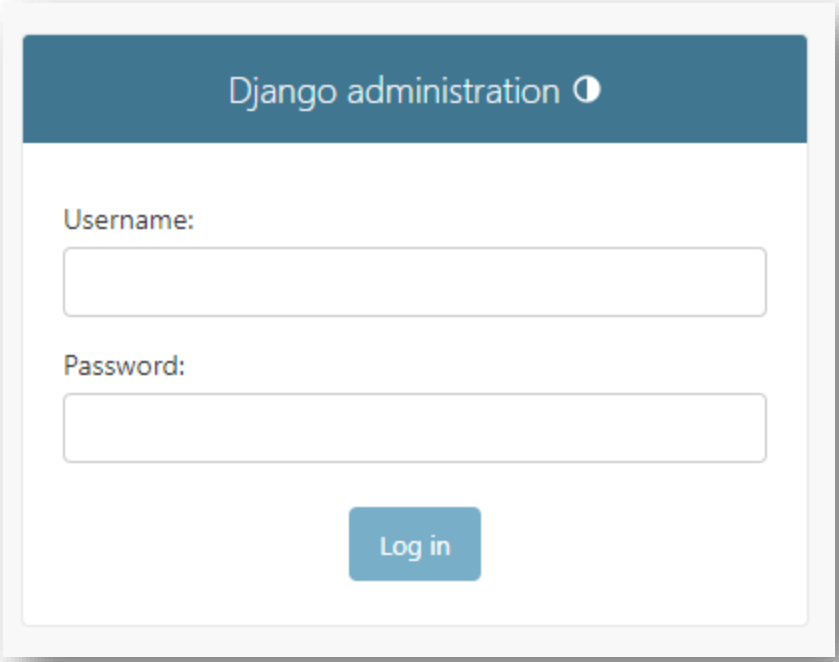
Also, please remember the password that you have entered. My password is: testuser@111

After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

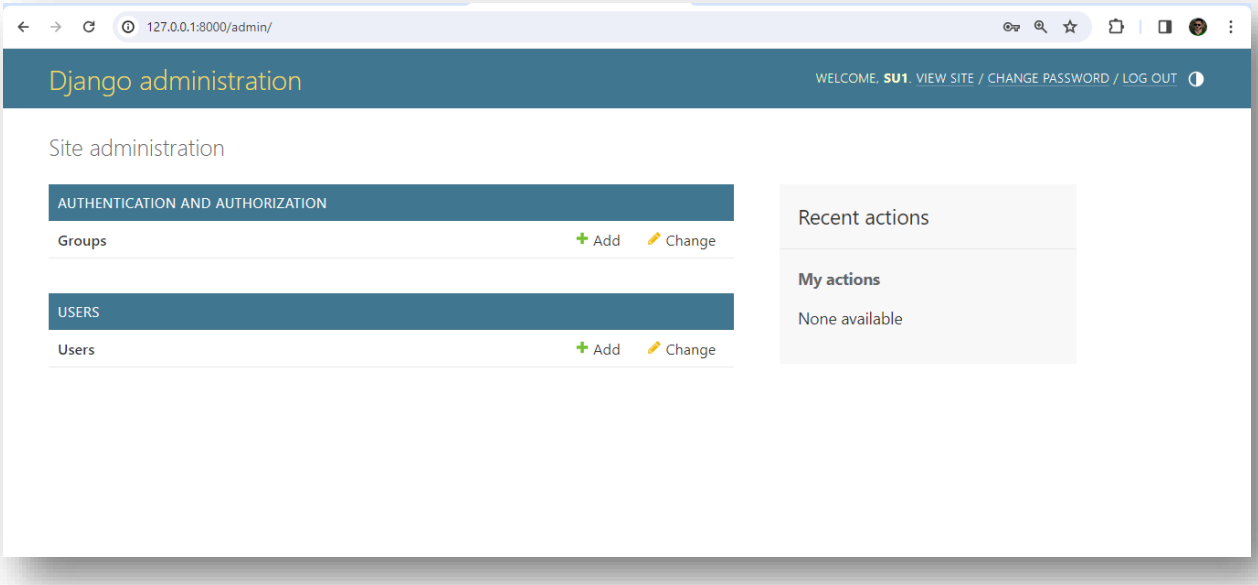
***command prompt***

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

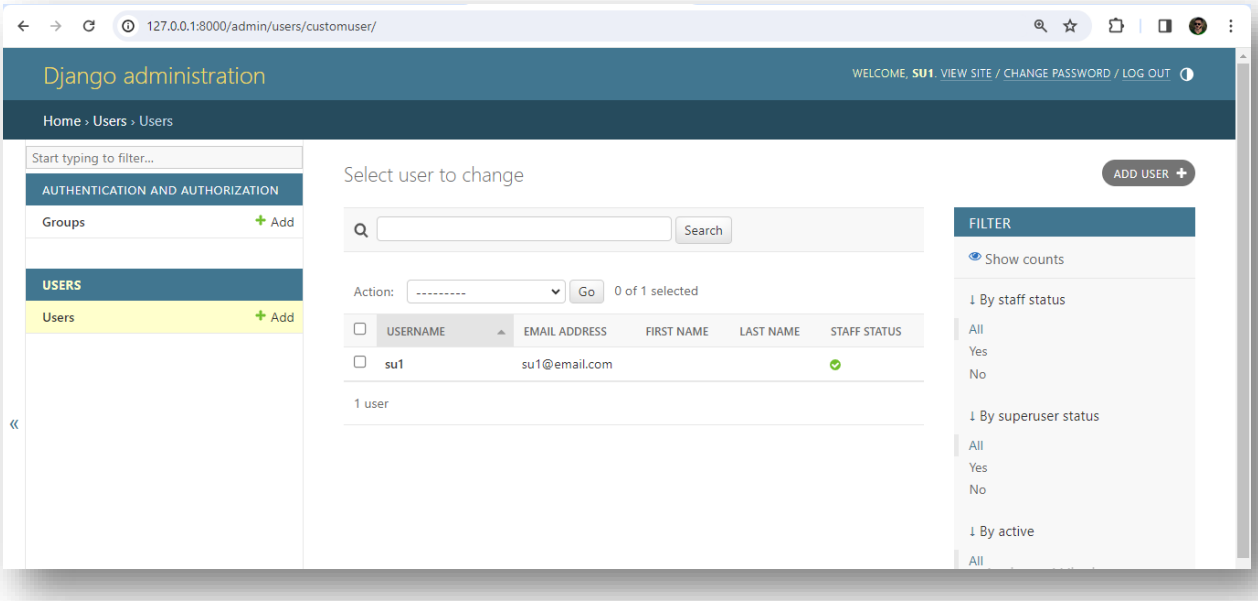
To log in as a superuser, visit: http://127.0.0.1:8000/admin/
You will see a login form, please enter the username and password you have just created here.




You will see something like this there.




If we click on the users tab. We will see the username, email address, first name, last name, and staff status.

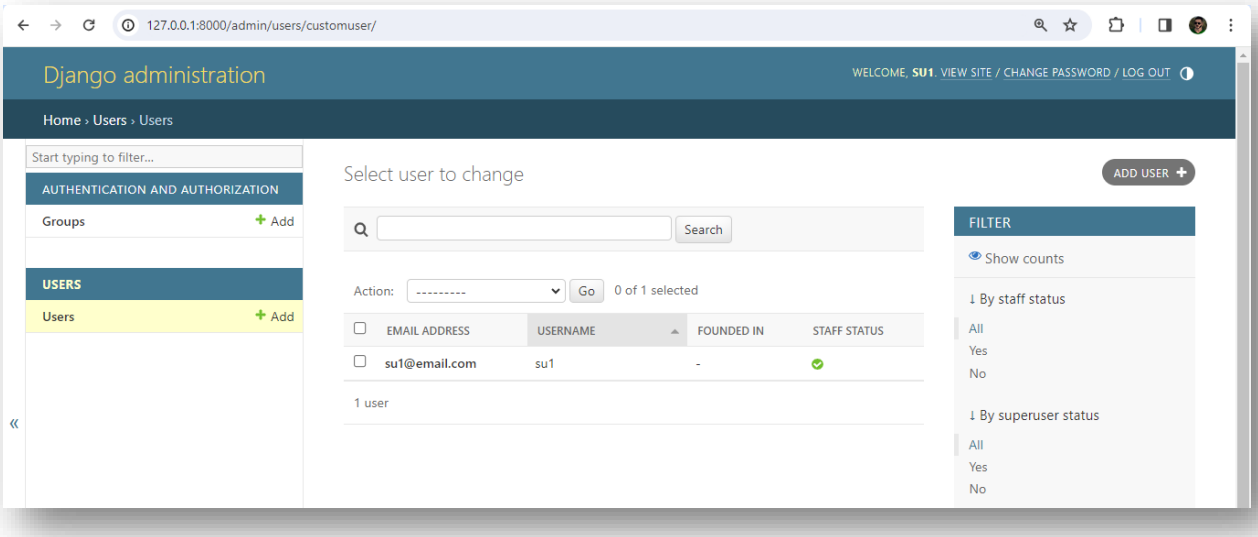



But we can customize this panel using the list_display. Let's see how:

***users/admin.py***

```python
# Import necessary modules from Django
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin

# Import forms and models defined in the same directory
from .forms import CustomUserCreationForm, CustomUserChangeForm
from .models import CustomUser

# Create a custom admin class for our CustomUser model, inheriting from UserAdmin
class CustomUserAdmin(UserAdmin):
    # Specify the form used for adding a new user
    add_form = CustomUserCreationForm
    # Specify the form used for updating an existing user
    form = CustomUserChangeForm
    # Specify the model for which this admin class is being defined
    model = CustomUser
    # Define the fields to be displayed in the list view of the admin page
    list_display = ['email', 'username', 'founded_in', 'is_staff', ]

# Register the CustomUser model with its corresponding admin class
admin.site.register(CustomUser, CustomUserAdmin)
```

Now, save your file and refresh the page.




Now you can see the updated fields.

## Conclusion

In conclusion, this chapter has equipped you with the essential knowledge to create a custom user model in Django. By exploring the reasons behind customization and providing a step-by-step guide, you now have the tools to design authentication systems tailored to your project's needs. This newfound understanding will be crucial as you continue to build robust and user-centric web applications using Django.

# Chapter 9: User Authentication

Having successfully implemented a functional custom user model, we can now incorporate essential functionalities that every website requires: user sign-up, login, and logout.

While Django offers built-in support for login and logout processes, we'll be required to develop our own form for user registration. Additionally, we'll construct a straightforward homepage featuring links to all three features, eliminating the need to manually input URLs each time.

Now let's create the template files.

***command prompt***

```
> mkdir templates
> echo>  templates/base.html
> echo>  templates/home.html
> echo>   templates/signup.html
> mkdir templates\registration
> echo> templates\registration\login.html
```

Now let's update the settings.py

***settings.py***

```
from pathlib import Path
import os


TEMPLATES = [
    {
        …
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        …
    },
]


LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home'
```

The code sets up a list of template directories for a Django project, specifying a directory path based on the project's base directory. It also defines login and logout redirect URLs as 'home'.

***base.html***

```
{% extends 'base.html' %}
```

```
{% block title %}
  Home
{% endblock title %}

{% block content %}
  {% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}!</p>
    <p><a href="{% url 'logout' %}">Log Out</a></p>
  {% else %}
    <p>You are not logged in</p>
    <p><a href="{% url 'login' %}">Log In</a> | <a href="{% url 'signup' %}">Sign
Up</a></p>
  {% endif %}
{% endblock content %}
```

This code displays a personalized welcome message and logout link if the user is authenticated; otherwise, it shows a message and login/signup links.

***base.html***

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JobVista</title>
</head>
<body>
<main>
<h1>JobVista</h1>
{% block content %}
{% endblock content %}
</main>
</body>
</html>
```

This HTML code defines a basic webpage titled "JobVista" with a main heading, and it includes a placeholder for content within the {% block content %} and {% endblock content %} tags

***templates/registration/login.html***

```
{% extends 'base.html' %}
{% block content %}
<h2>Log In</h2>
<form method="post">
{% csrf_token %}
{{ form.as_p }}
<button type="submit">Log In</button>
```

```
</form>
{% endblock content %}
```

This Django template code defines a block named "content" that contains an HTML form for user login. The form uses the POST method, includes a CSRF token, and displays form fields (username and password) using the {{ form.as_p }} template tag.

***templates/signup.html***

```
{% extends 'base.html' %}
{% block title %}Sign Up{% endblock title %}
{% block content %}
<h2>Sign Up</h2>
<form method="post">
{% csrf_token %}
{{ form.as_p }}
<button type="submit">Sign Up</button>
</form>
{% endblock content %}
```

This code represents a Django template block for rendering a sign-up form. It includes an HTML form with a POST method, a CSRF token, and a button to submit the form, utilizing a Django form object (form) to render the form fields.

Now let's create urls.py.

***command prompt***

```
> echo> users/urls.py
```

Now let's update the urls.py file.

***users/urls.py***

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

urlpatterns = [
path('signup/', SignUpView.as_view(), name='signup'),
path('logout/', views.logoutUser, name="logout"),
]
```

This code defines URL patterns for two views: one for user registration at the 'signup/' endpoint using the SignUpView class, and another for user logout at the 'logout/' endpoint using the logoutUser function from the views module.

Now let's update *JobVista/urls.py.*

***JobVista/urls.py***

```
from django.contrib import admin
from django.urls import path, include
from django.views.generic.base import TemplateView


urlpatterns = [
path('admin/', admin.site.urls),
path('users/', include('users.urls')),
path('users/', include('django.contrib.auth.urls')),
path('', TemplateView.as_view(template_name='home.html'),
name='home'),
]
```

This Django code defines URL patterns for a web application: it includes an admin interface at '/admin/', user-related URLs from 'users.urls', authentication URLs from 'django.contrib.auth.urls', and a default home view using a template named 'home.html' at the root URL.

***users/views.py***

```
# Import necessary modules and classes from Django
from django.urls import reverse_lazy
from django.views.generic import CreateView
from .forms import CustomUserCreationForm
from django.contrib.auth import logout
from django.shortcuts import redirect

# Define a class called SignUpView that inherits from Django's CreateView class
class SignUpView(CreateView):
    # Specify the form class to be used for user registration
    form_class = CustomUserCreationForm

    # Set the URL to redirect to upon successful user registration (uses
reverse_lazy for deferred URL resolution)
    success_url = reverse_lazy('login')

    # Specify the template (HTML file) to be used for rendering the signup page
    template_name = 'signup.html'
```

```python
# Define a function called logoutUser that logs out the current user and
redirects them to the 'home' URL
def logoutUser(request):
    # Use Django's logout function to log out the current user
    logout(request)

    # Redirect the user to the 'home' URL after logging out
    return redirect('home')
```

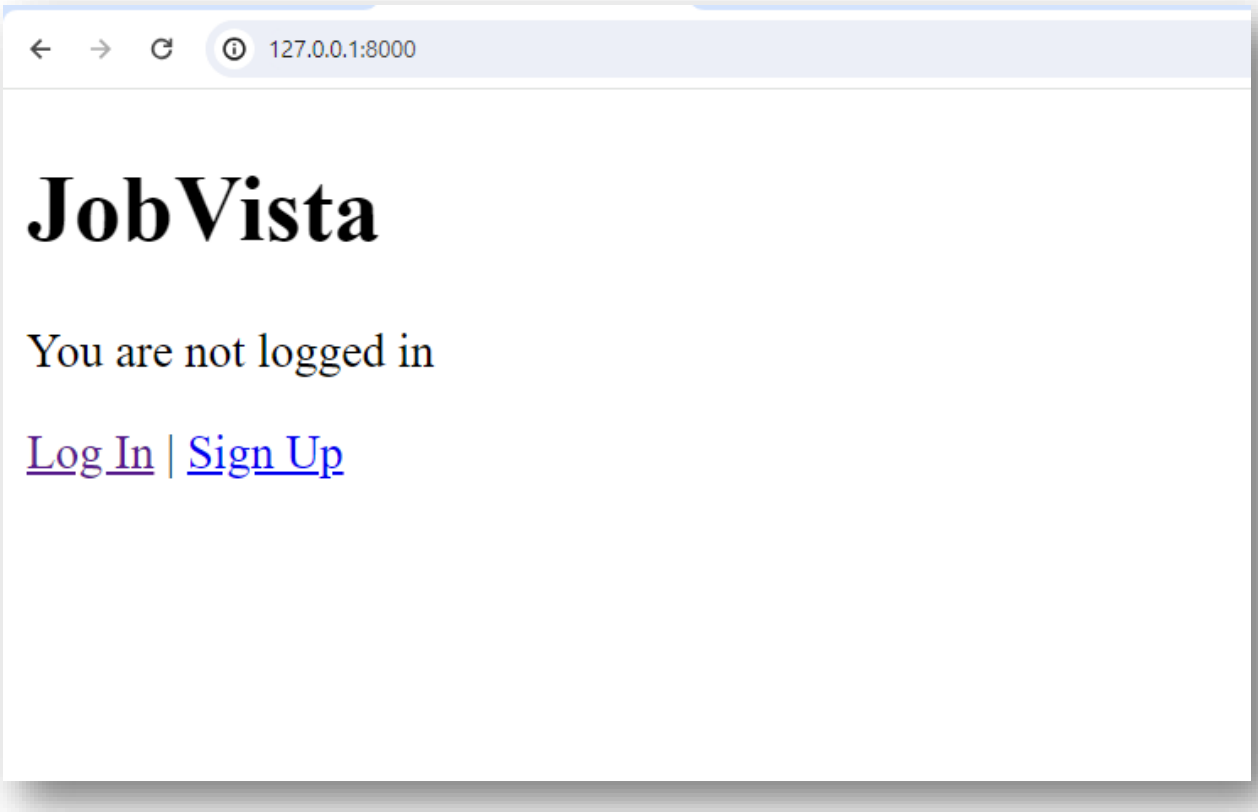
After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

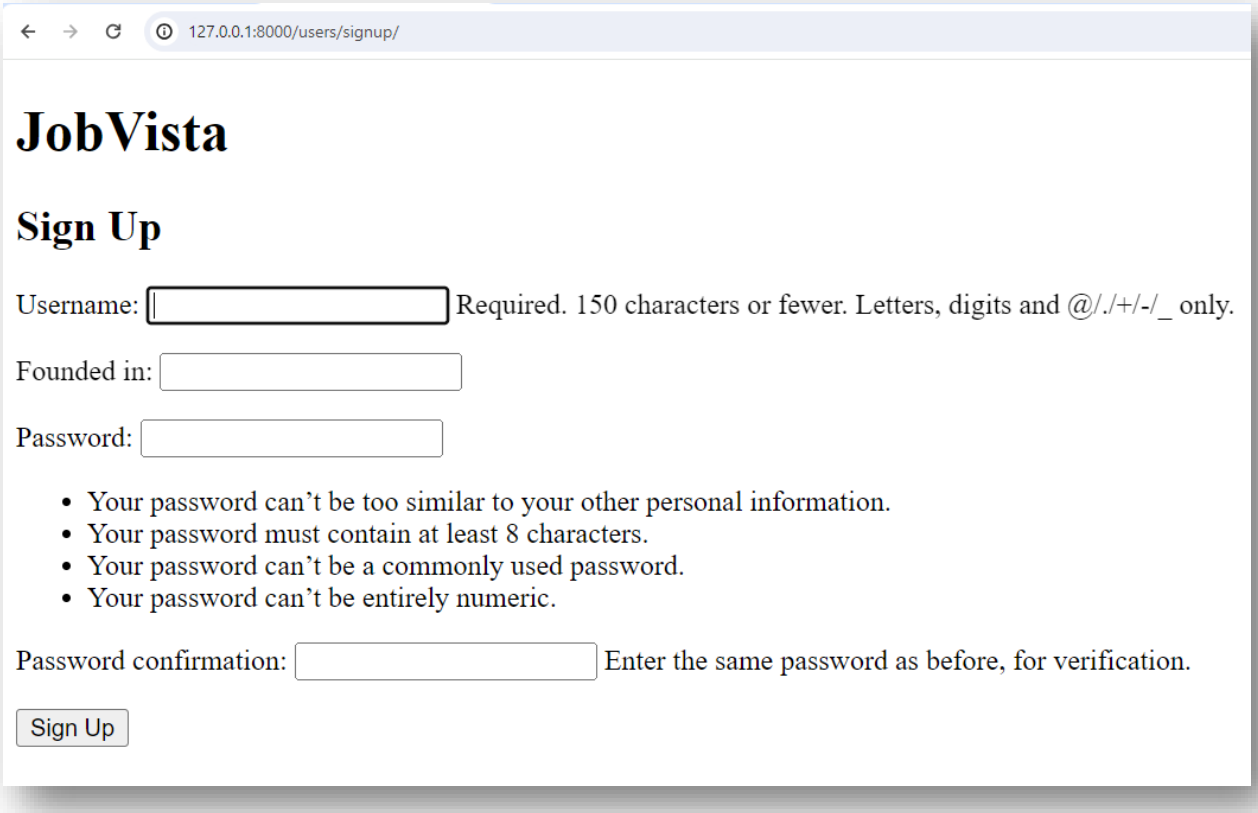
```
> python manage.py runserver
```

Visit: http://127.0.0.1:8000/


Let's log out, by clicking on the log out button.



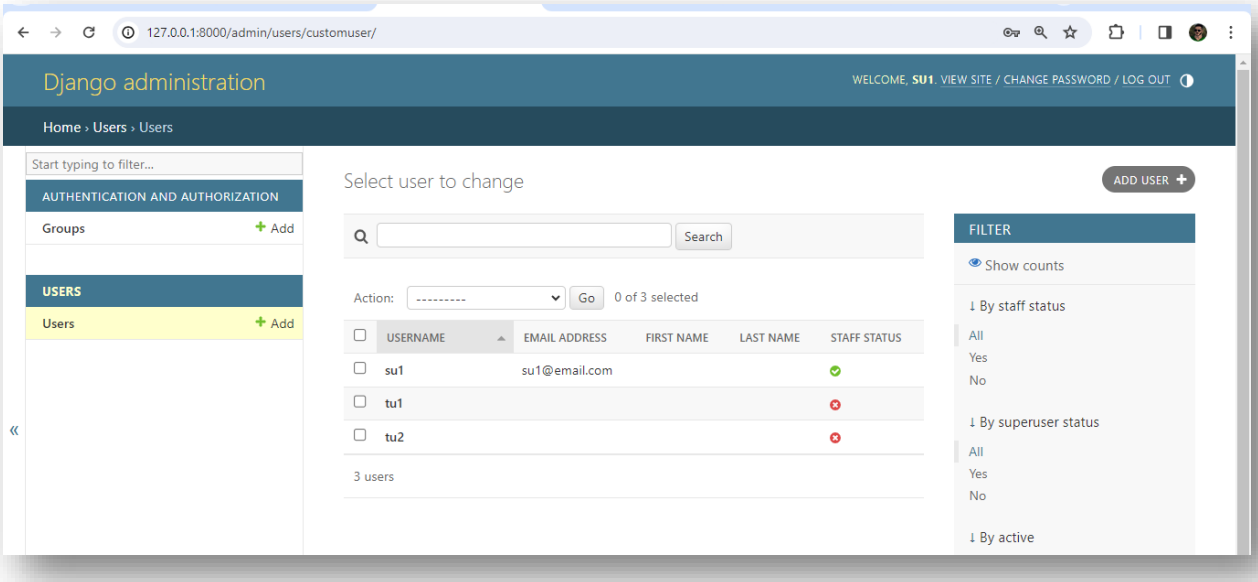

Now let’s get registered as a new user. To do that click on Sign Up and fill in the details.




Now let's log in with a newly created password.


And you will get the login homepage.


Now let’s see our admin page.




Now let’s make our website a little more beautiful.

## Conclusion

In summary, the User Authentication chapter lays the groundwork for secure and efficient user identity management in Django. We explored key components like the user model, authentication views, and forms. By leveraging Django's built-in features and emphasizing DRY principles, developers can establish a robust authentication system. This knowledge serves as a crucial foundation for upcoming chapters, where we'll delve into advanced features while prioritizing a seamless user experience.

# Chapter 10: Bootstrap

## Pages

Currently, our website consists of a single page known as the home page, and our sole app is named "news." However, as we continue to enhance our website, it becomes imperative to have specialized apps catering to various functionalities within our projects.

To facilitate this, let's establish a dedicated app called "pages" and transfer our homepage from the users' app to the new "pages" app.

***command prompt***

```
> python manage.py startapp pages
```

Now Let's work on the setting.py and add the newly created app there.

***settings.py***

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'users',
    'pages',
]
```

Let's make changes to JobVista/urls.py. In this step, we'll eliminate the homepage URL and incorporate the URLs for various pages.

**JobVista/urls.py**

```
from django.contrib import admin
from django.urls import path, include
from django.views.generic.base import TemplateView


urlpatterns = [
path('admin/', admin.site.urls),
path('users/', include('users.urls')),
path('users/', include('django.contrib.auth.urls')),
path('', include('pages.urls')),
]
```

*command prompt*

```
> echo> pages/urls.py
```

Now let's update pages/urls.py

**pages/urls.py**

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

urlpatterns = [
path('', HomePageView.as_view(), name='home'),
]
```

This code defines a URL pattern for the root path ('') that maps to the HomePageView class-based view and assigns the name 'home' to this URL pattern.

Let's update the pages/views.py

**pages/views.py**

```
from django.shortcuts import render
from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = 'home.html'
```

This code defines a view class, HomePageView, using the generic TemplateView, and sets the template to 'home.html' for rendering the home page.

## Tests

Now let's create tests for our updated application.

Occasionally, incorrect Python code indentation may be observed due to constraints in the page width. To resolve this issue, please copy the code and paste it into a text editor such as Visual Studio Code (VS Code). This should help ensure proper indentation.

**news/tests.py**

```
from django.test import TestCase

from django.contrib.auth import get_user_model
from django.test import SimpleTestCase, TestCase
from django.urls import reverse
from django.test import TestCase, Client
```

```
from .views import HomePageView

class HomePageTests(SimpleTestCase):
    def setUp(self):
        self.client = Client()

    def test_home_page_status_code(self):
        # Ensure the home page returns a successful status code
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_view_url_by_name(self):
        # Ensure the URL for the home page is accessible by name
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_view_uses_correct_template(self):
        # Ensure the home page uses the correct HTML template
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'home.html')

class SignupPageTests(TestCase):
    def setUp(self):
        self.client = Client()
        # Creative username and email for testing
        self.username = 'fantasyUser123'
        self.email = 'fantasyuser@email.com'

    def test_signup_page_status_code(self):
        # Ensure the signup page returns a successful status code
        response = self.client.get('/users/signup/')
        self.assertEqual(response.status_code, 200)

    def test_view_url_by_name(self):
        # Ensure the URL for the signup page is accessible by name
        response = self.client.get(reverse('signup'))
        self.assertEqual(response.status_code, 200)

    def test_view_uses_correct_template(self):
        # Ensure the signup page uses the correct HTML template
        response = self.client.get(reverse('signup'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, 'signup.html')
```

```
    def test_signup_form(self):
        # Create a new user using the custom user model
        new_user = get_user_model().objects.create_user(
            self.username, self.email)

        # Ensure the new user is added to the database
        self.assertEqual(get_user_model().objects.all().count(), 1)

        # Ensure the username and email of the new user match the provided values
        self.assertEqual(get_user_model().objects.all()[0].username,
self.username)
        self.assertEqual(get_user_model().objects.all()[0].email, self.email)
```

This tests.py file includes test cases for string representation, content, list view, detail view, create view, update view, and delete view for the Article model and views in your Django app.

To execute the tests, stop the server using Control+c, and enter the command "python manage.py test" in the command line:

***command prompt***

```
> python manage.py test
```

You will get a similar output as the following:
Output:

***command prompt***

```
Found 7 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.......
----------------------------------------------------------------------
Ran 7 tests in 0.366s

OK
Destroying test database for alias 'default'...
```

## Bootstrap

Bootstrap, an influential front-end framework, enables developers to build responsive and visually appealing websites swiftly. It provides an extensive collection of pre-designed HTML, CSS, and JavaScript components, including navigation bars, buttons, forms, and more, effectively streamlining the development process.

Here's how to seamlessly integrate Bootstrap into your Django projects:

**Include Bootstrap:**
We can effortlessly integrate Bootstrap by linking to its Content Delivery Network (CDN) within our HTML templates. Alternatively, we can download Bootstrap and include the local files directly within your project.

**Template Integration:**
We can utilize Bootstrap features by incorporating its CSS classes and JavaScript components into our Django templates.We can build well-structured layouts with Bootstrap's renowned grid system.We can style buttons, forms, and other elements with ease using Bootstrap's extensive styling options.

**Customization:**
We can customize Bootstrap's default styles to align perfectly with our project's unique design.We can achieve this customization by overriding Bootstrap's styles within our project's CSS file.

**Django Integration Packages:**
We can use packages designed specifically for Bootstrap and Django collaboration, such as django-bootstrap4 or django-crispy-forms. These packages provide handy template tags and utilities, simplifying Bootstrap's integration into your Django forms and templates.

For a comprehensive exploration of Bootstrap's features, visit the official documentation: Bootstrap Documentation: https://getbootstrap.com/docs/5.2/

Now let's use the bootstrap inside our project.
Let's update home.html

**templates/home.html**

```
{% extends 'base.html' %}

{% block title %}
  Home
{% endblock title %}

{% block content %}
  {% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}!</p>

  {% else %}
    <p>You are not logged in.</p>

  {% endif %}
{% endblock content %}
```

This code extends a base HTML file and defines two blocks: "title" and "content." The content block displays a welcome message if the user is authenticated, otherwise, it indicates that the user is not logged in.

Let's update base.html and connect it with bootstrap cdn and urls.

**templates/base.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-
to-fit=no">
    <title>JobVista</title>
    <!-- Add Bootstrap CSS link -->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
    <style>
        body {
            background-color: #f8f9fa; /* Set a light background color for the
body */
        }

        .navbar {
            background-color: #007bff; /* Set the background color for the navbar
*/
        }

        .navbar-brand, .navbar-nav .nav-link {
            color: white !important; /* Set text color to white for navbar links
and brand */
        }

        .navbar-nav .nav-item.active {
            background-color: #0056b3 !important; /* Set the background color for
active link */
        }
    </style>
</head>
<body>

<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <a class="navbar-brand" href="{% url 'home' %}">JobVista</a>
```

```
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav ml-auto">
            <li class="nav-item {% if request.path == '/login/' %}active{% endif
%}">
                <a class="nav-link" href="{% url 'login' %}">Log In</a>
            </li>
            <li class="nav-item {% if request.path == '/signup/' %}active{% endif
%}">
                <a class="nav-link" href="{% url 'signup' %}">Sign Up</a>
            </li>
            {% if user.is_authenticated %}
                <li class="nav-item">
                    <a class="nav-link" href="{% url 'logout' %}">Log Out</a>
                </li>
            {% endif %}
        </ul>
    </div>
</nav>

<main class="container mt-4">
    {% block content %}
    {% endblock content %}
</main>

<!-- Add Bootstrap JS and Popper.js scripts -->
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"><
/script>
<script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></sc
ript>

</body>
</html>
```

Now let's visit the homepage and see how it looks.
Visit: http://127.0.0.1:8000/

## Conclusion

In this chapter, we explored the symbiotic relationship between Django and Bootstrap, unlocking the potential to seamlessly integrate style and responsiveness into our web applications. We've gained the tools to strike a harmonious balance between aesthetics and functionality. As we navigate our Django journey, the knowledge gained here will be a cornerstone for creating dynamic, modern, and visually captivating web applications.

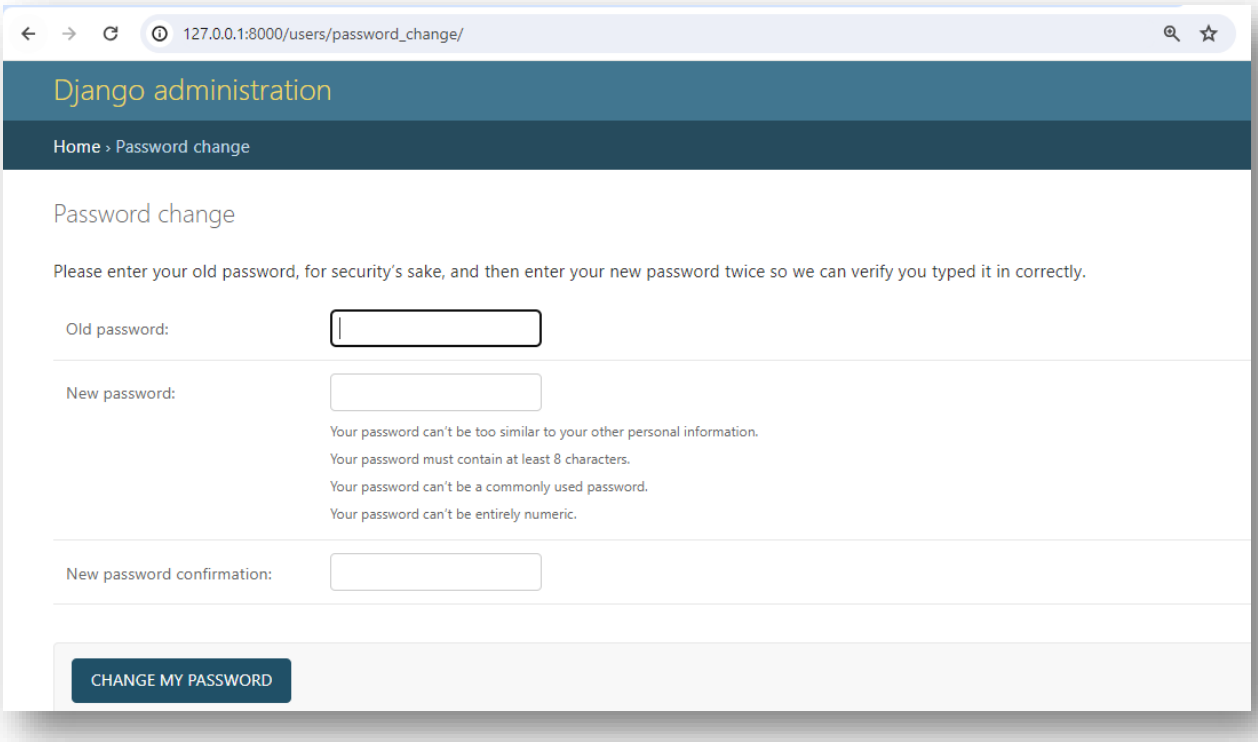
# Chapter 11: Password Change and Reset

In this chapter, we will improve the authorization process of our JobVista app by incorporating functionalities for changing and resetting passwords.

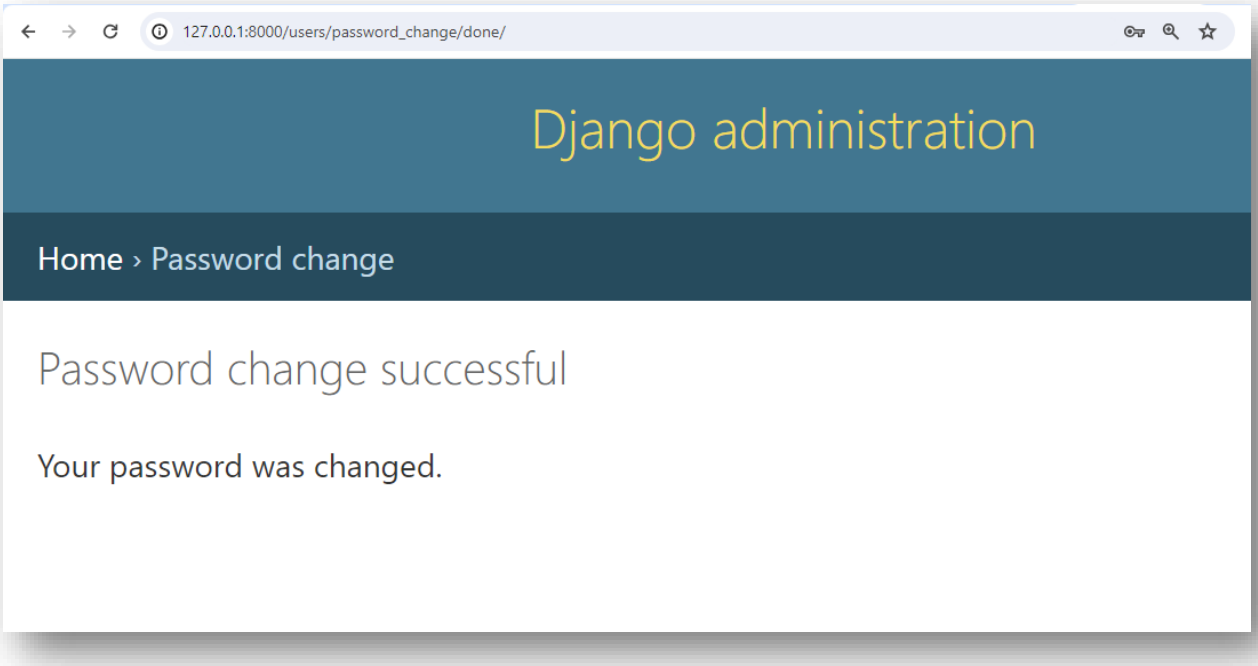
Users have the option to either update their existing password or initiate a password reset through email if it has been forgotten. Django comes with pre-built views and URLs for these operations. Initially, we'll examine the default implementations before tailoring them to our needs using Bootstrap-based templates and an email service.

To test the password change feature, click on "Log In," verify that you are logged in, and then navigate to http://127.0.0.1:8000/users/password_change/.

| You must be logged in to change the password. |
|---|




Let's change our password to testuser@222
Visit: http://127.0.0.1:8000/users/password_change/done/




The password has been updated. Currently, the password can only be changed through the admin interface. Let's establish dedicated pages for this purpose

***command prompt***

```
> echo> templates/registration/password_change_form.html
> echo>templates/registration/password_change_done.html
```

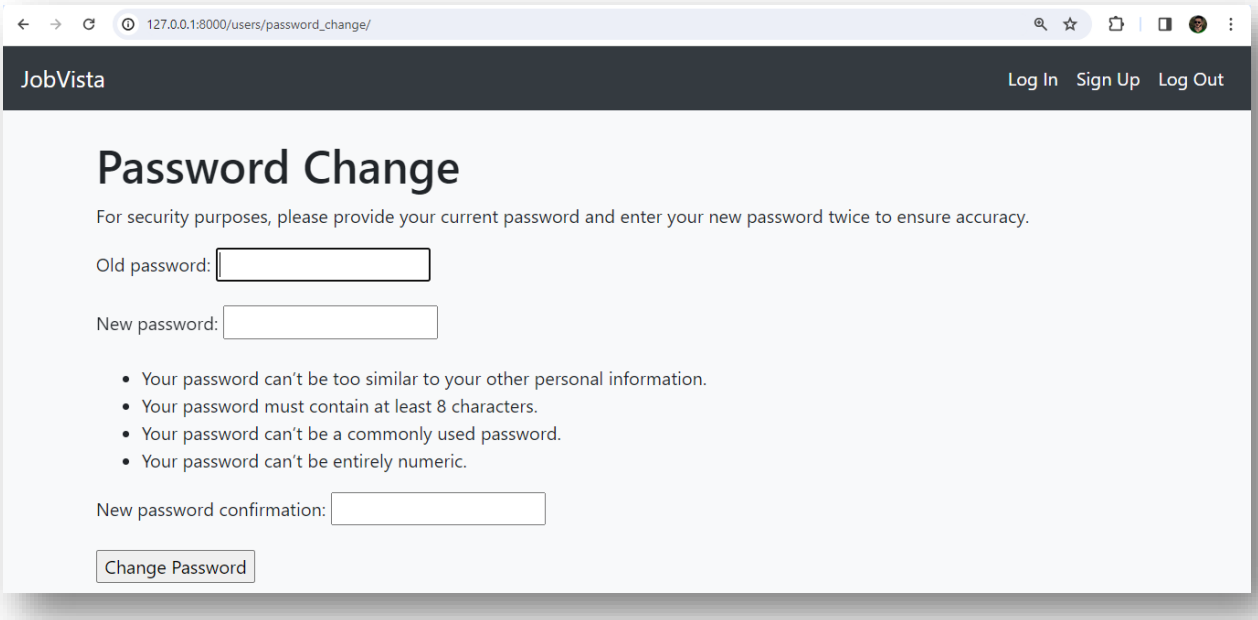
Let’s update the password_change_form.html

**templates/registration/password_change_form.html**

```
{% extends 'base.html' %}
{% block title %}Change Password{% endblock title %}
{% block content %}
  <h1>Password Change</h1>
  <p>For security purposes, please provide your current password and enter your
new password twice to ensure accuracy.</p>
  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Change Password">
  </form>
{% endblock content %}
```

This HTML code displays a "Password Change" heading and a paragraph instructing users to provide their current password and enter a new password twice for security. It includes a form with CSRF token protection and renders a password change form with a submit button.

Visit: http://127.0.0.1:8000/users/password_change/.



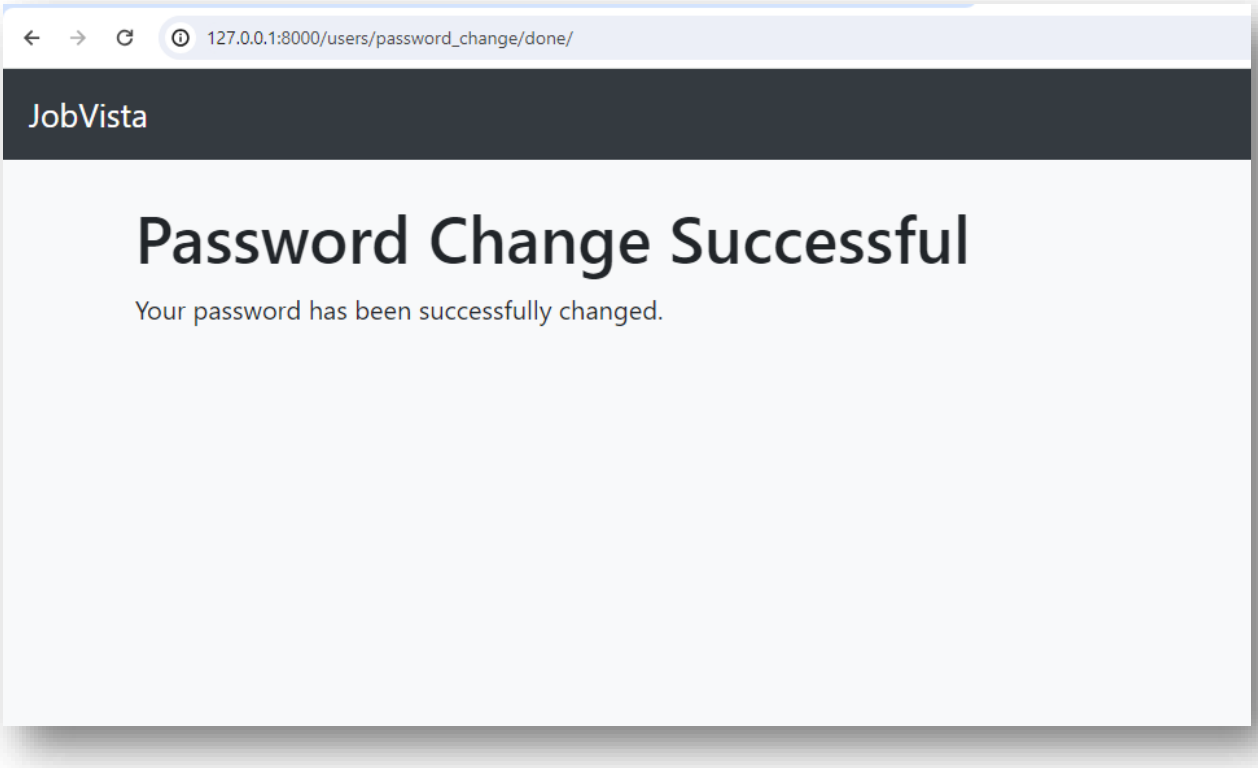

Let's update the password_change_done.html

**templates/registration/password_change_done.html**

```
{% extends 'base.html' %}
{% block title %}Password Change Successful{% endblock title %}
{% block content %}
  <h1>Password Change Successful</h1>
  <p>Your password has been successfully changed.</p>
{% endblock content %}
```

This code extends a base HTML template and defines blocks for the title and content. In this specific instance, it sets the title to "Password Change Successful" and includes a header and paragraph in the content block to indicate a successful password change.

Visit: http://127.0.0.1:8000/users/password_change/done/




## Password reset

Many websites incorporate a password reset feature to assist users who may forget their passwords. To initiate the password reset process, you can proceed through the following steps:

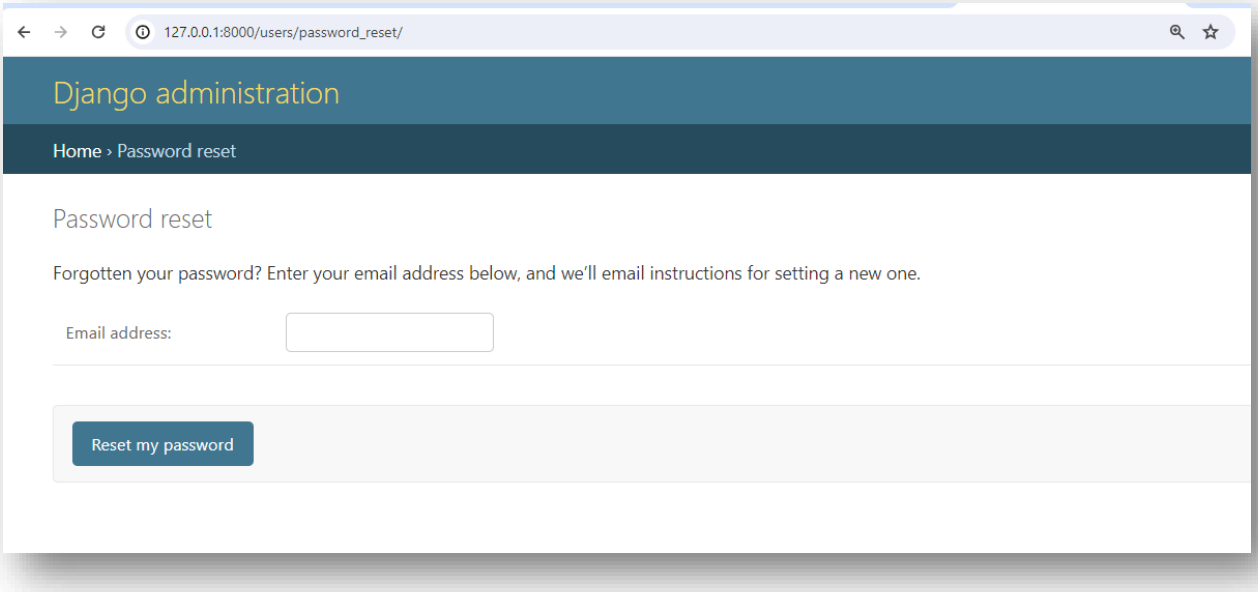
- Utilize Django's default implementation for password reset.
- Customize templates to match the visual aesthetics of your website.
- Configure Django to send emails—In a production setting, SendGrid can be employed, while for testing purposes, Django's console backend is suitable.

To implement these changes, open the settings.py file and append the following text at the end:

***settings.py***

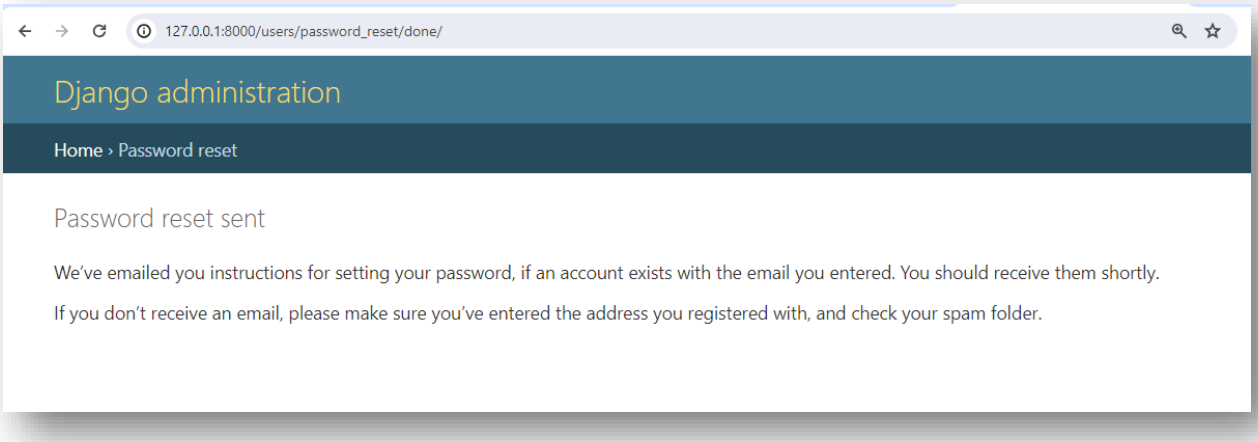
```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Django handles the rest.
Now visit: http://127.0.0.1:8000/users/password_reset/




Ensure that the email you provide corresponds to one of your user accounts. After submission, you will be redirected to the password reset completion page at

http://127.0.0.1:8000/users/password_reset/done/.



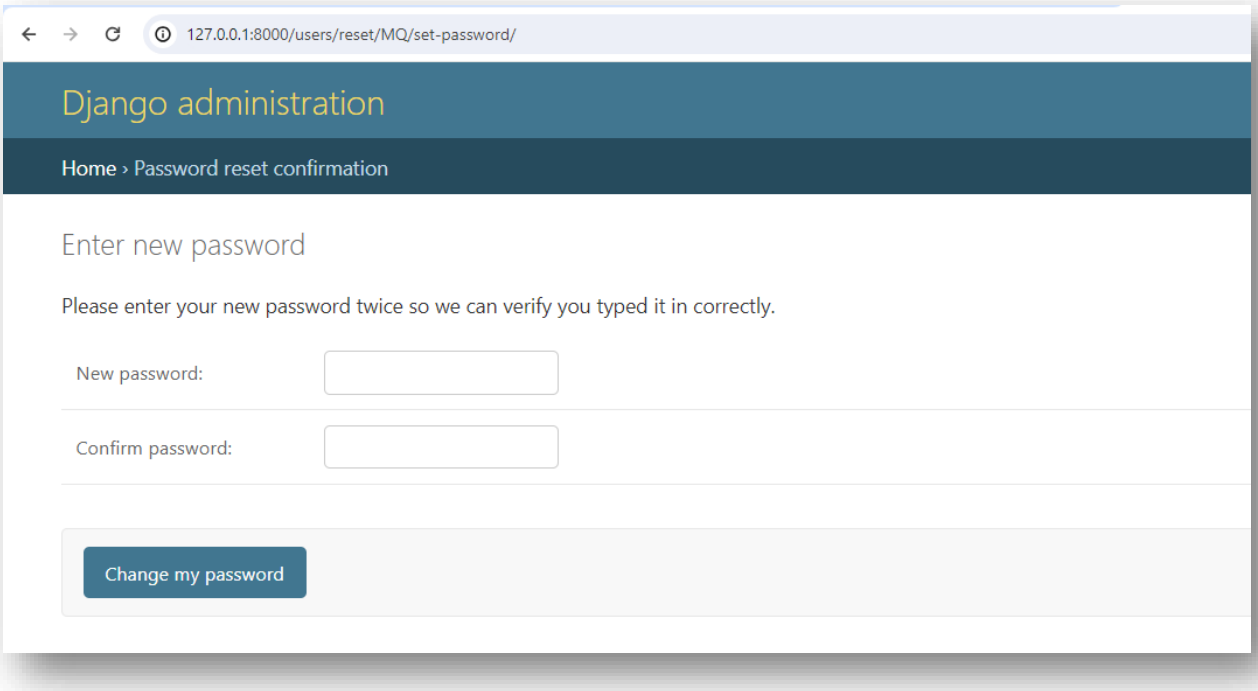

You will see a text similar to the following text:

***command prompt***

```
Django version 5.0, using settings 'JobVista.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

[11/Dec/2023 03:23:51] "GET /users/password_reset/ HTTP/1.1" 200 3421
Content-Type: text/plain; charset="utf-8"
```

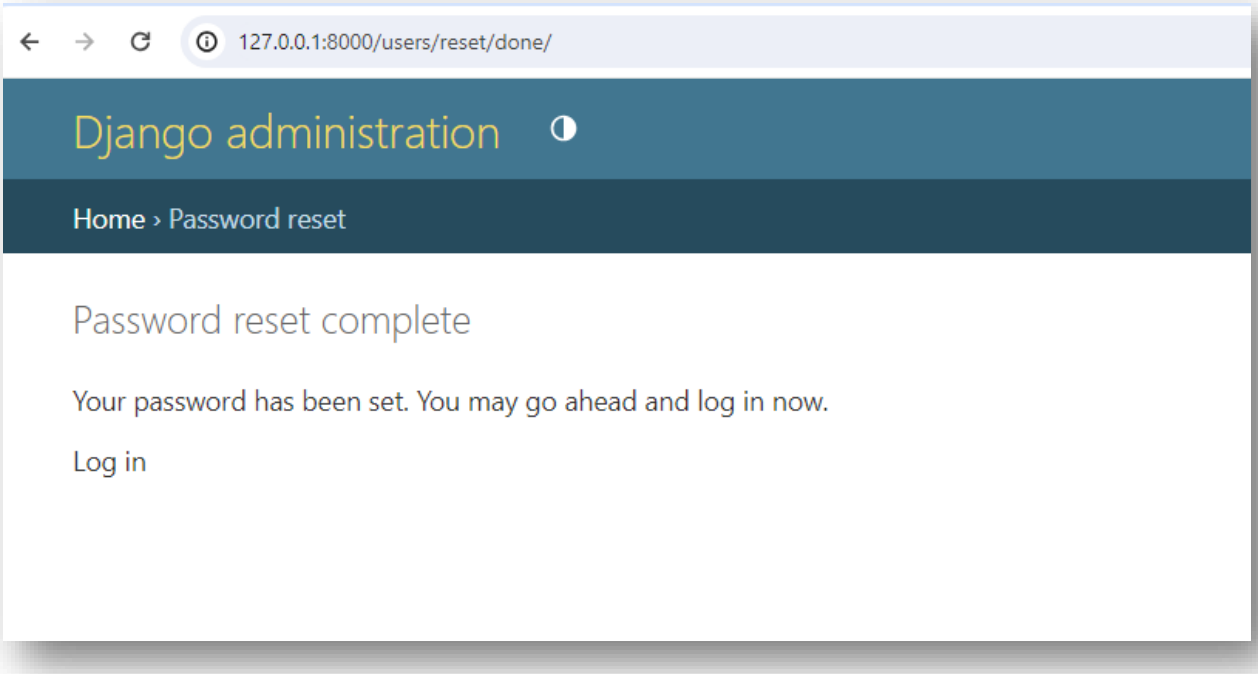
```
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: su1@email.com
Date: Sun, 10 Dec 2023 21:57:16 -0000
Message-ID: <170224543658.11412.26283483500238825@DESKTOP-4EI07JU>


You're receiving this email because you requested a password reset for your user account at
127.0.0.1:8000.

Please go to the following page and choose a new password:

http://127.0.0.1:8000/users/reset/MQ/bz0xlg-6a2a4228e422405a38bcf12422bbc139/

Your username, in case you've forgotten: su1

Thanks for using our site!

The 127.0.0.1:8000 team



----------------------------------------------------------------------
[11/Dec/2023 03:27:16] "POST /users/password_reset/ HTTP/1.1" 302 0
[11/Dec/2023 03:27:16] "GET /users/password_reset/done/ HTTP/1.1" 200 2991
```

Follow the link provided to reset the password.It looks similar to:
http://127.0.0.1:8000/users/reset/MQ/bz0xlg-6a2a4228e422405a38bcf12422bbc139/




Let's change it to testuser@111
You will see the following page.

http://127.0.0.1:8000/users/reset/done/




Now let's create custom HTML pages for these pages.

***command prompt***

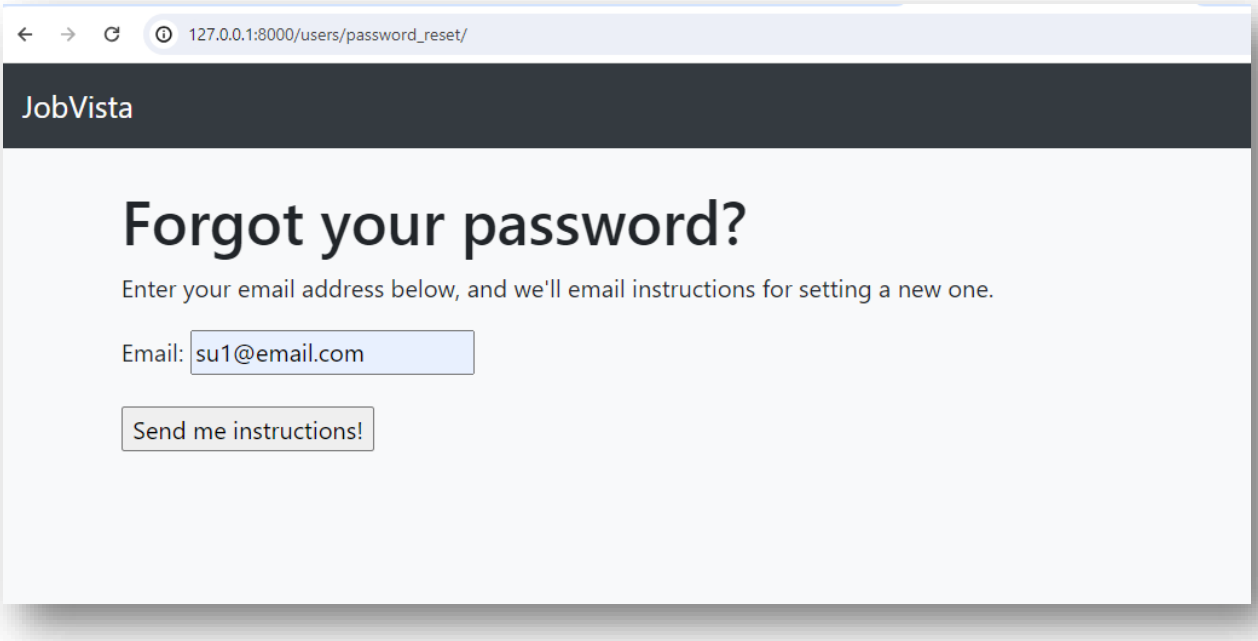
```
> echo> templates/registration/password_reset_form.html
> echo> templates/registration/ password_reset_done.html
> echo> templates/registration/password_reset_confirm.html
>echo> templates/registration/password_reset_complete.html
```

Let's update **password_reset_form.html**

**templates/registration/password_reset_form.html**

```
{% extends 'base.html' %}
{% block title %}Forgot Your Password?{% endblock %}
{% block content %}
  <h1>Forgot your password?</h1>
  <p>Enter your email address below, and we'll email instructions for setting a
new one.</p>
  <form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <input  type="submit" value="Send me instructions!">
  </form>
{% endblock %}
```

You can test it here: http://127.0.0.1:8000/users/password_reset/

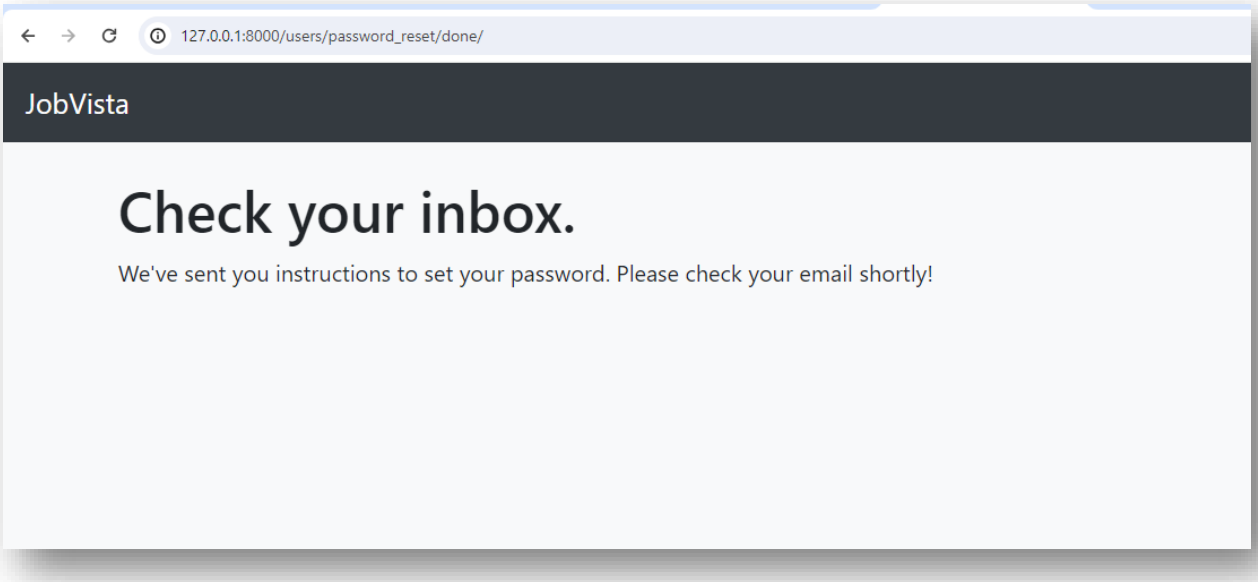



Let's update password_reset_done.html

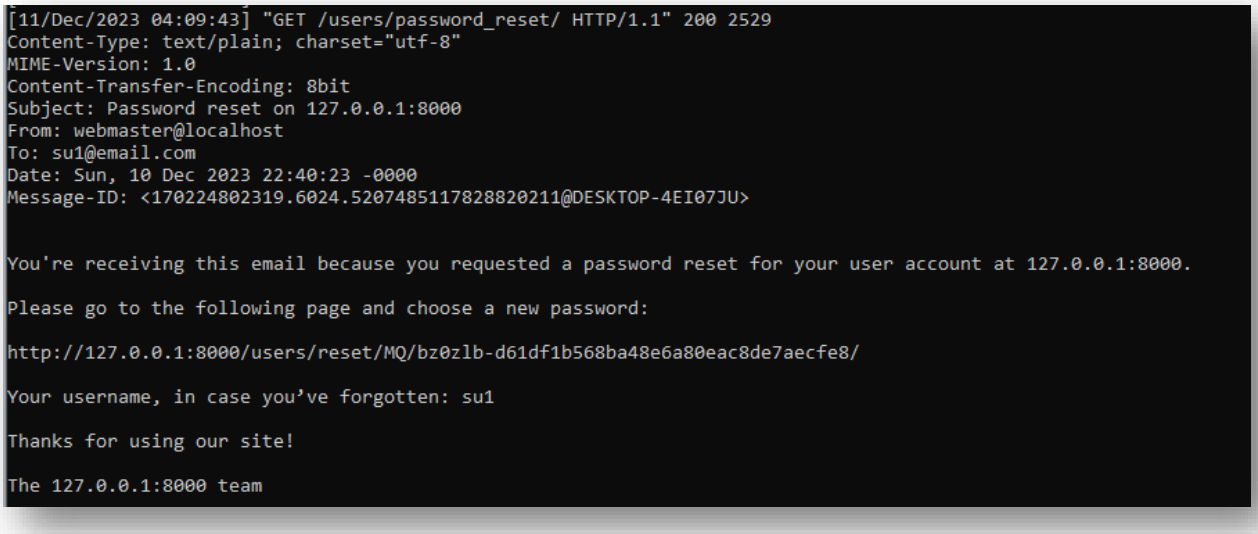
**templates/registration/password_reset_done.html**

```
{% extends 'base.html' %}
{% block title %}Email Confirmation{% endblock title %}
{% block content %}
<h1>Check your inbox.</h1>
<p>We've sent you instructions to set your password. Please check your email
shortly!</p>
{% endblock content %}
```

Visit: http://127.0.0.1:8000/users/password_reset/done/.




Inside the command prompt:



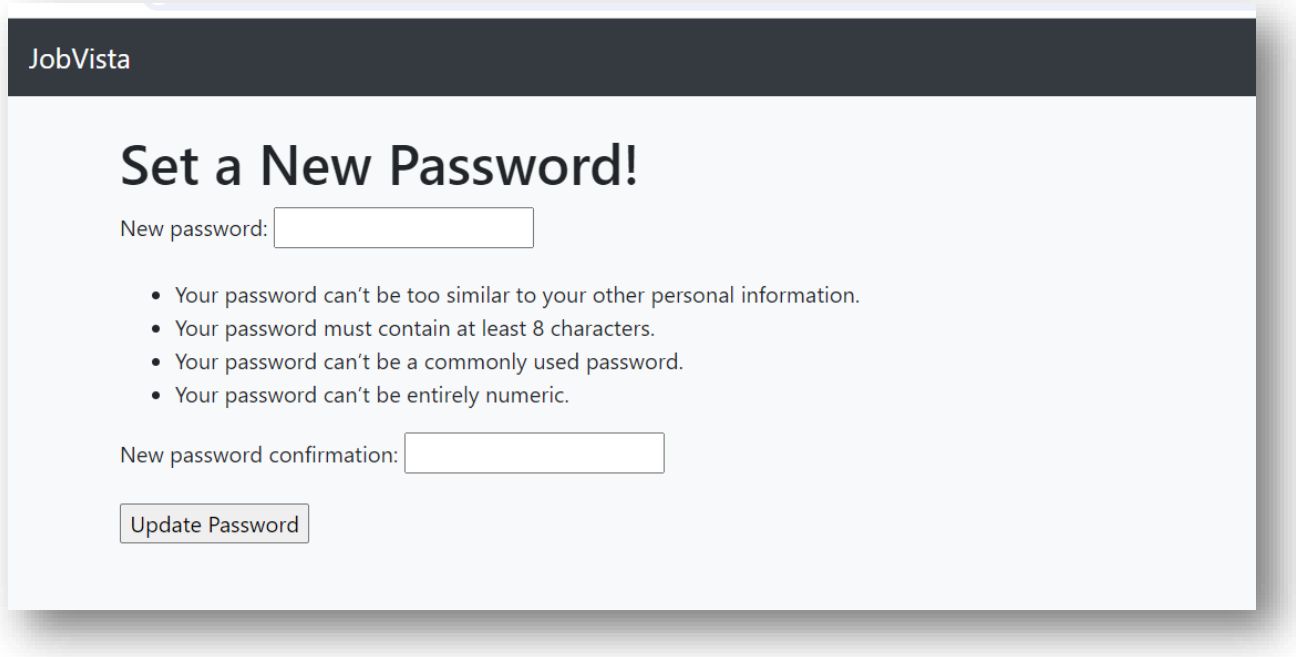

Let's update password_reset_confirm.html

**templates/registration/password_reset_confirm.html**

```
{% extends 'base.html' %}
{% block title %}Create a New Password{% endblock title %}
{% block content %}
<h1>Set a New Password!</h1>
<form method="POST">
{% csrf_token %}
{{ form.as_p }}
```

```
<input type="submit" value="Update Password">
</form>
{% endblock content %}
```

Get the link from the command prompt and Visit the link which is similar to:
http://127.0.0.1:8000/users/reset/MQ/bz0zlb-d61df1b568ba48e6a80eac8de7aecfe8/

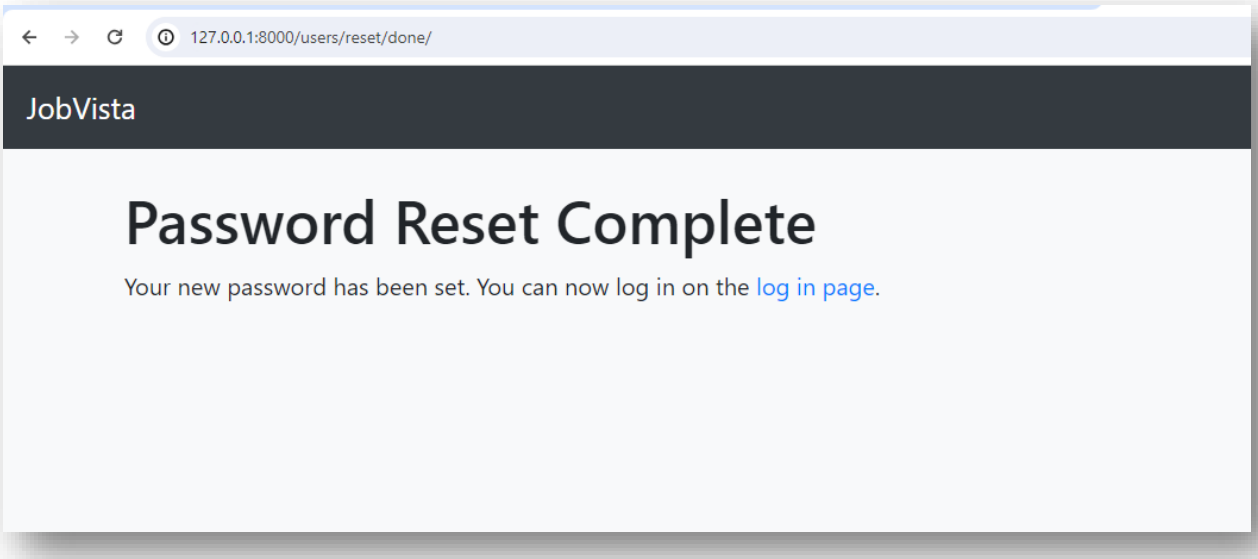



Let's update password_reset_complete.html

**templates/registration/password_reset_complete.html**

```
{% extends 'base.html' %}
{% block title %}Password Reset Complete{% endblock title %}
{% block content %}
<h1>Password Reset Complete</h1>
<p>Your new password has been set. You can now log in on the <a href="{% url
'login' %}">log in page</a>.</p>
{% endblock content %}
```

Visit: http://127.0.0.1:8000/users/reset/done/




## Conclusion

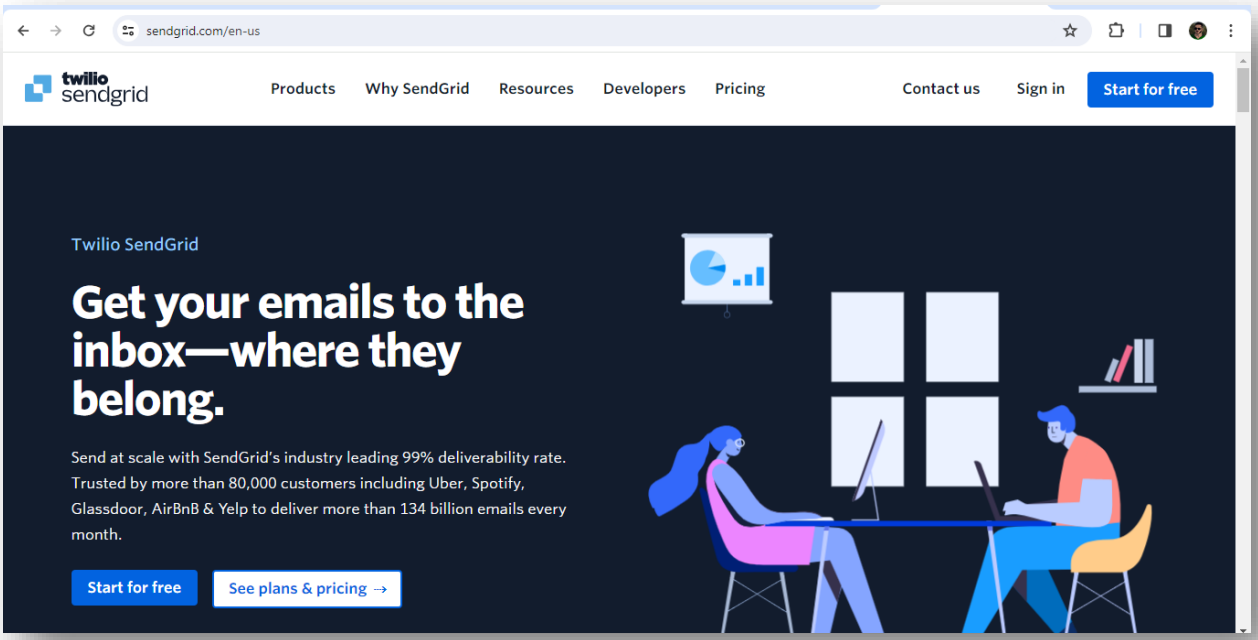
In this chapter, we focused on implementing password change and reset functionalities in our Django project. Exploring Django's authentication system and leveraging built-in views, we enabled users to seamlessly modify passwords and recover account access. As we conclude, we've not only enhanced our application's security but also deepened our understanding of the interplay between user authentication and application functionality. This newfound knowledge will be instrumental as we progress in building a robust and user-friendly web application with Django.

# Chapter 12: Email

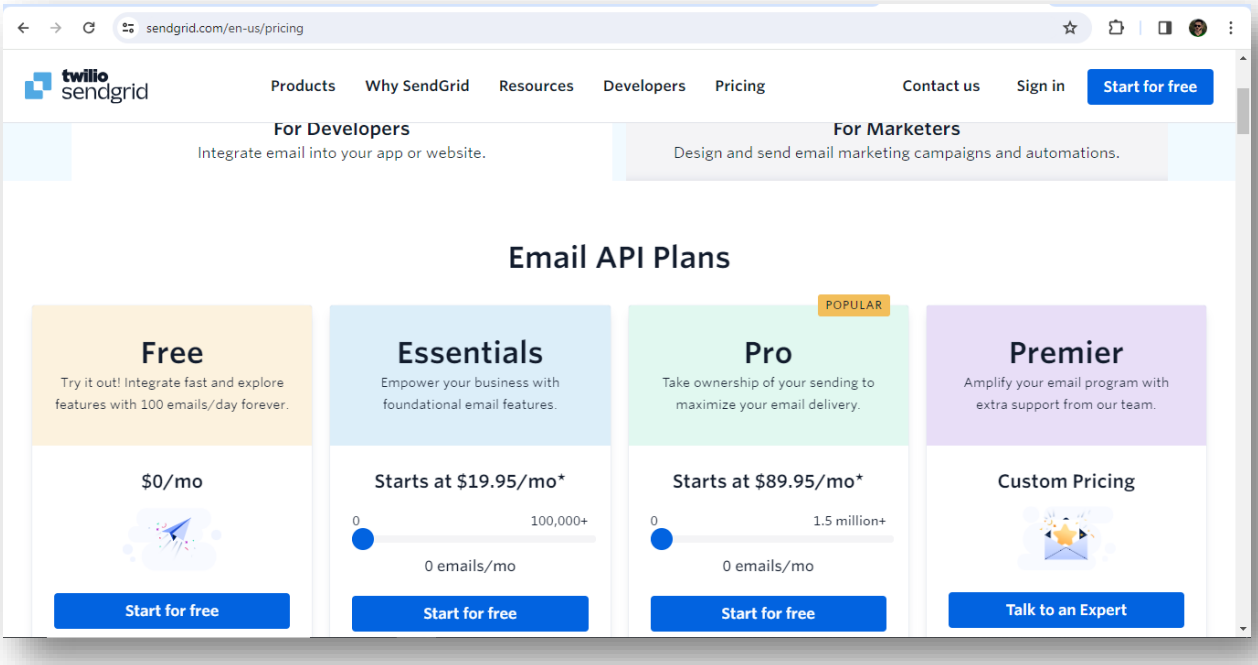
In this section, we'll establish a connection between SendGrid and our Django project, enabling us to send real emails rather than relying on the command prompt for password reset links.

SendGrid serves as a cloud-based email service, offering a platform for the delivery of both transactional and marketing emails. While other options such as Mailchimp or Amazon SES are available, we've opted for SendGrid in this particular project.

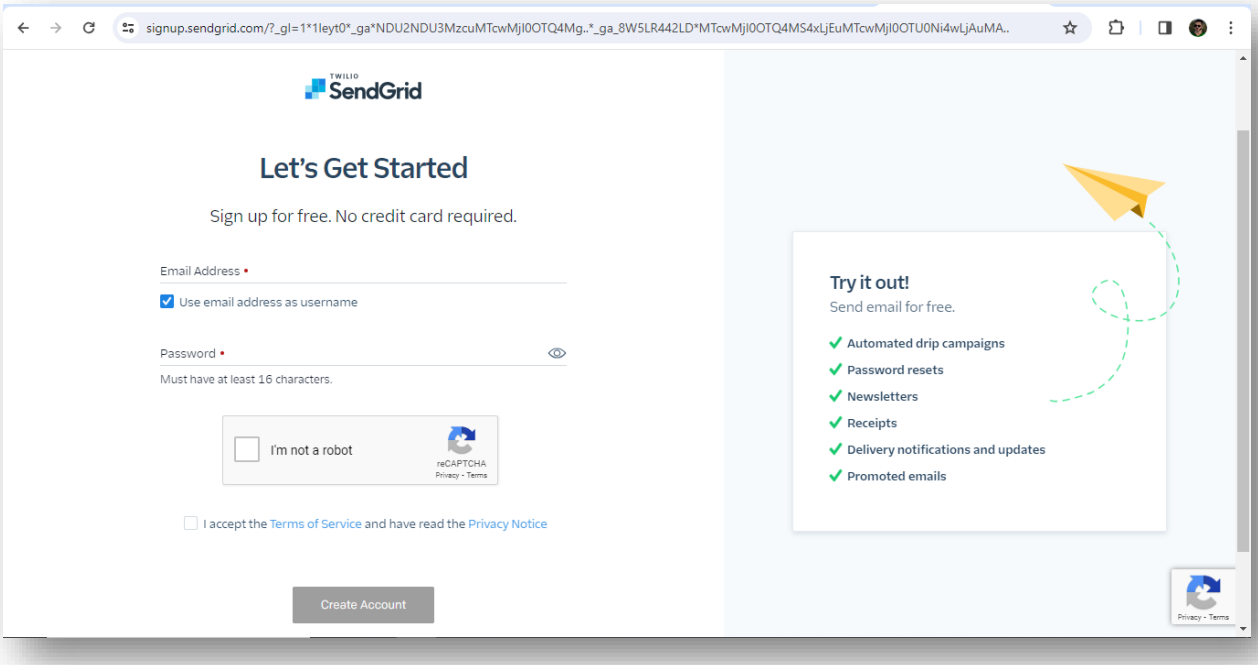
Notably, SendGrid's services are complimentary for up to 100 emails per day. Now let’s visit: https://sendgrid.com/




Visit the pricing page to access comprehensive details on our pricing structure.

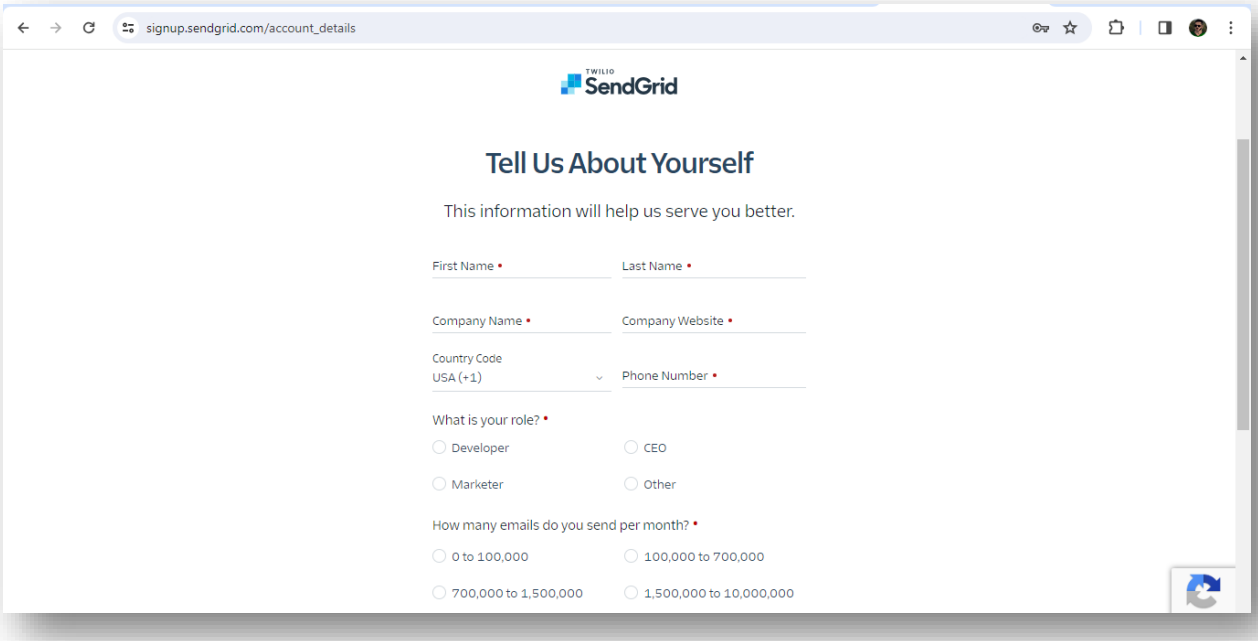



For the time being, opt for the free option and click the "Start with Free" button.

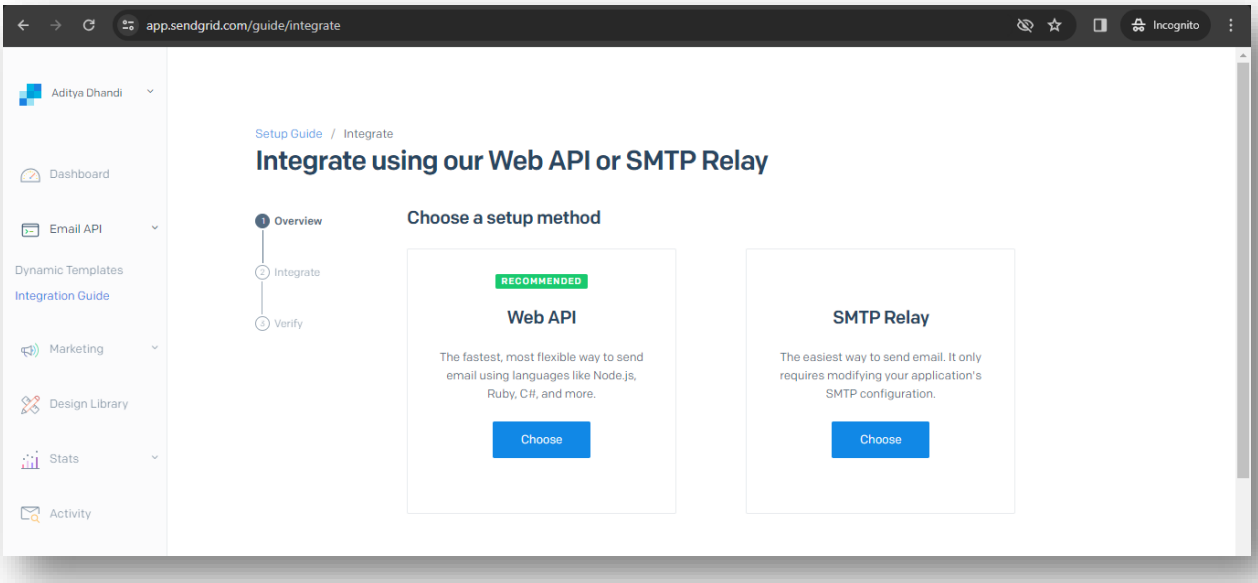



Please provide your signup details to create your account.

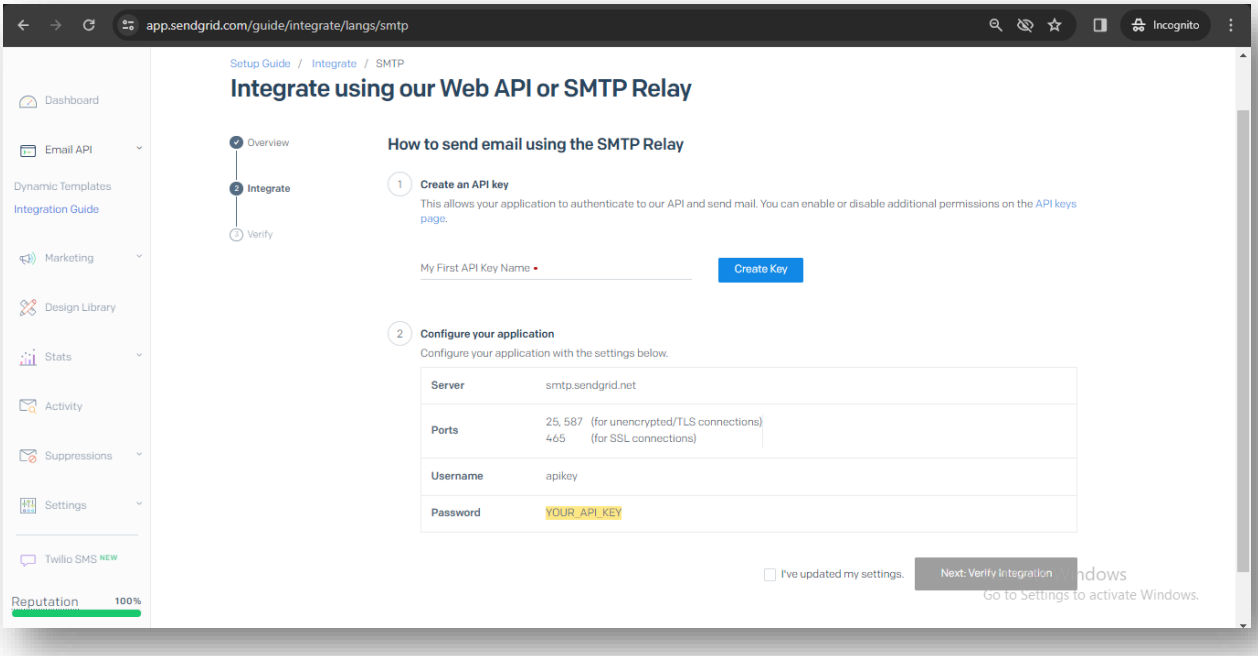
Kindly share the necessary information about yourself to proceed with the account creation process.




After logging in, navigate to the Email APIs section and locate the Integration Guide. From there, opt for the SMTP relay option.




Now, you need to assign a name to your API key. Choose any name you prefer.

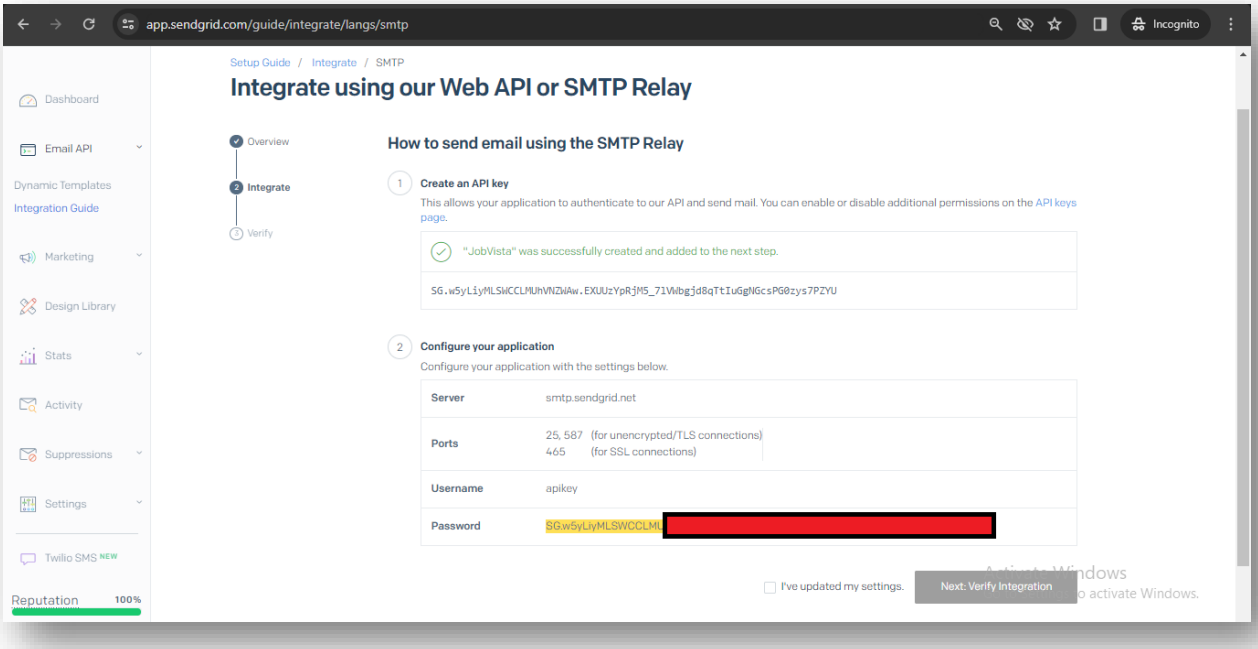



You will receive your ports, username, and password.Visit Settings.py in your project and paste the following code at the bottom of your file. Additionally, update the previously defined variable EMAIL_BACKEND with the provided information.

**settings.py**

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend' #updated

EMAIL_HOST = 'smtp.sendgrid.net'
EMAIL_HOST_USER = 'apikey'
EMAIL_HOST_PASSWORD = 'your password'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Replace 'your password' with your actual password. Now, check the "I have updated my settings" box and click on the "Next: Verify Integration" button.

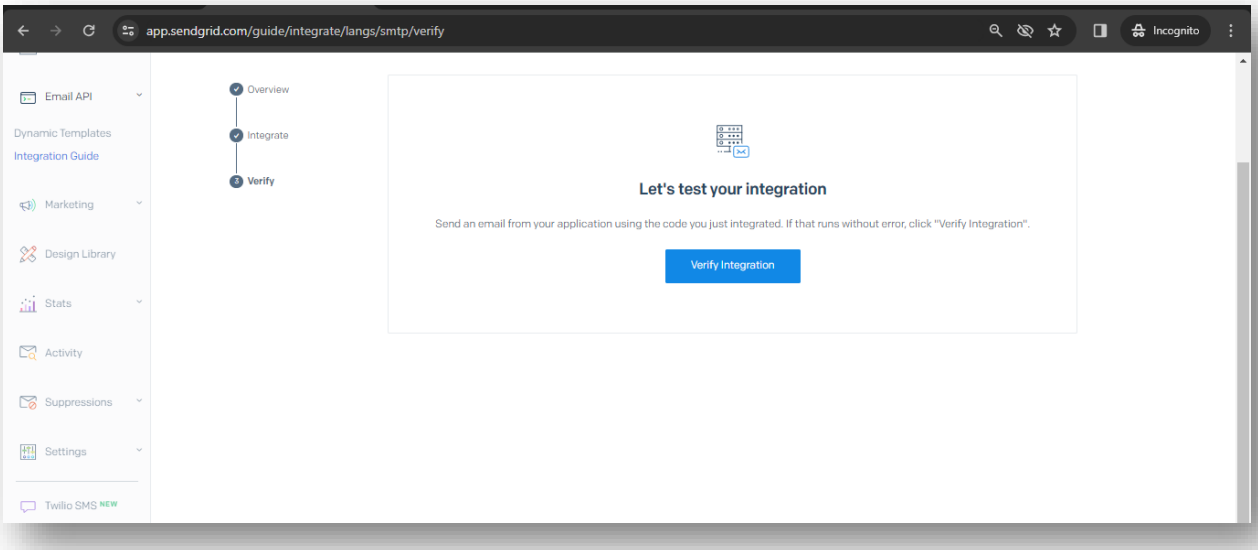



Visit the password reset form on your web browser.
http://127.0.0.1:8000/users/password_reset/

Enter the email address associated with your superuser account, ensuring it's distinct from the email used for your SendGrid account. Complete the form, and upon submission, you'll be redirected to:

http://127.0.0.1:8000/users/password_reset/done/

Now, go to your email inbox. Look for a new email from webmaster@localhost; its content should match what was displayed in our command line console earlier in the process. As a final step, return to the SendGrid page. Since we've successfully tested the application, click on the blue "Verify Integration" button.




If all of this works, congratulations! You have successfully integrated an email service into your project.

## Conclusion

In this chapter, we established a seamless connection between SendGrid and our local Django project, enhancing our application's email capabilities. This integration enables efficient email handling, scalability, and sets the stage for advanced features. As we conclude, we've successfully integrated SendGrid, laying the groundwork for a more dynamic user experience in the upcoming chapters.

# Chapter 13: The Job Listing App

Now, let's develop a fully functional job listing website with all the essential features. To achieve this, we need to create a new application called 'listings.' Now, Open your command prompt.

***command prompt***

```
> python manage.py startapp listings
```

Let’s update settings.py

**settings.py**

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'users',
    'pages',
    'listings',

]

TIME_ZONE = 'America/New_York'
```

This code establishes the installed apps, encompassing both standard modules and custom ones ('users', 'pages', 'listings'). Additionally, it configures the application's time zone to 'America/New_York'.

Next, we'll proceed to crafting the 'model.py' file for this application.

**listings/models.py**

```
# Import necessary modules from Django
from django.db import models
from django.conf import settings
from django.contrib.auth import get_user_model
from django.urls import reverse

# Define a Django model called Listing
class Listing(models.Model):
    # Define a field for the job title, limiting its length to 255 characters
```

```
    job_title = models.CharField(max_length=255)

    # Define a field for the job description, allowing for longer text
    description = models.TextField()

    # Define a field for the date, automatically set to the current date and time
when the object is created
    date = models.DateTimeField(auto_now_add=True)

    # Define a field for the company associated with the listing, using a
ForeignKey relationship to the User model
    company = models.ForeignKey(
        get_user_model(),  # Get the user model specified in Django settings
        on_delete=models.CASCADE,  # Specify that when a related user is deleted,
also delete the associated listings
    )

    # Define a human-readable representation of the object (used in admin and
debugging)
    def __str__(self):
        return self.job_title

    # Define a method to get the absolute URL for a detailed view of the listing
    def get_absolute_url(self):
        return reverse('listing_detail', args=[str(self.id)])
```

The **get_absolute_url** method defines how to get the absolute URL for a detailed view of a listing. It uses the reverse function to generate the URL based on the URL pattern named 'listing_detail' and the listing's ID.

In summary, this code defines a Django model for job listings, specifying fields like job title, description, date, and the associated company. It also provides methods for a human-readable representation and generating the absolute URL for detailed views.

Now, let's make the migrations.

***command prompt***

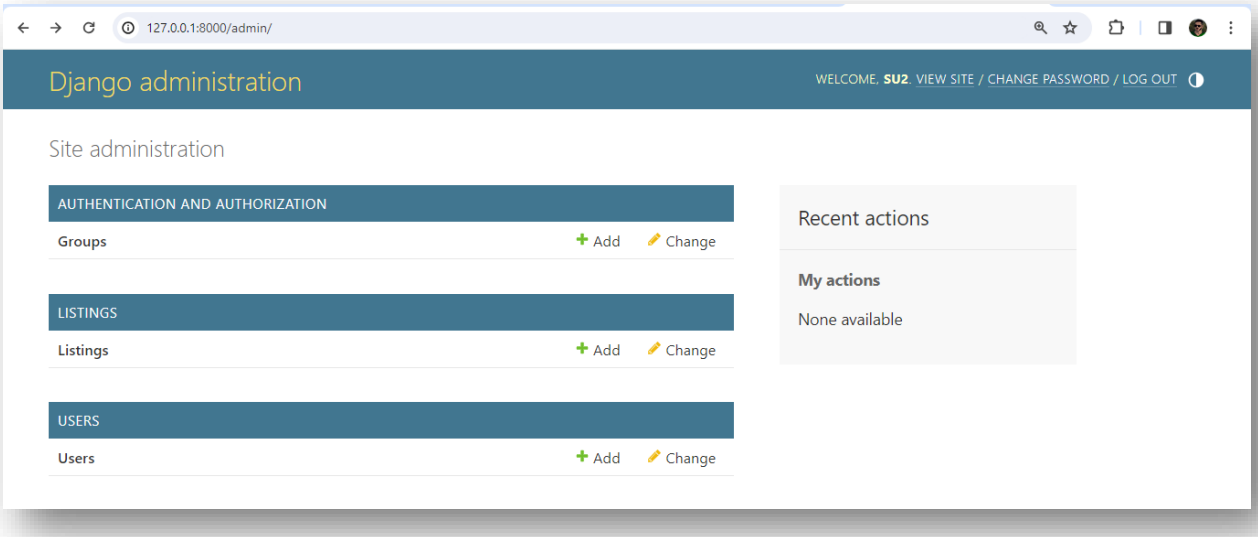
```
> python manage.py makemigrations users
> python manage.py migrate
```

Now let's update admin.py

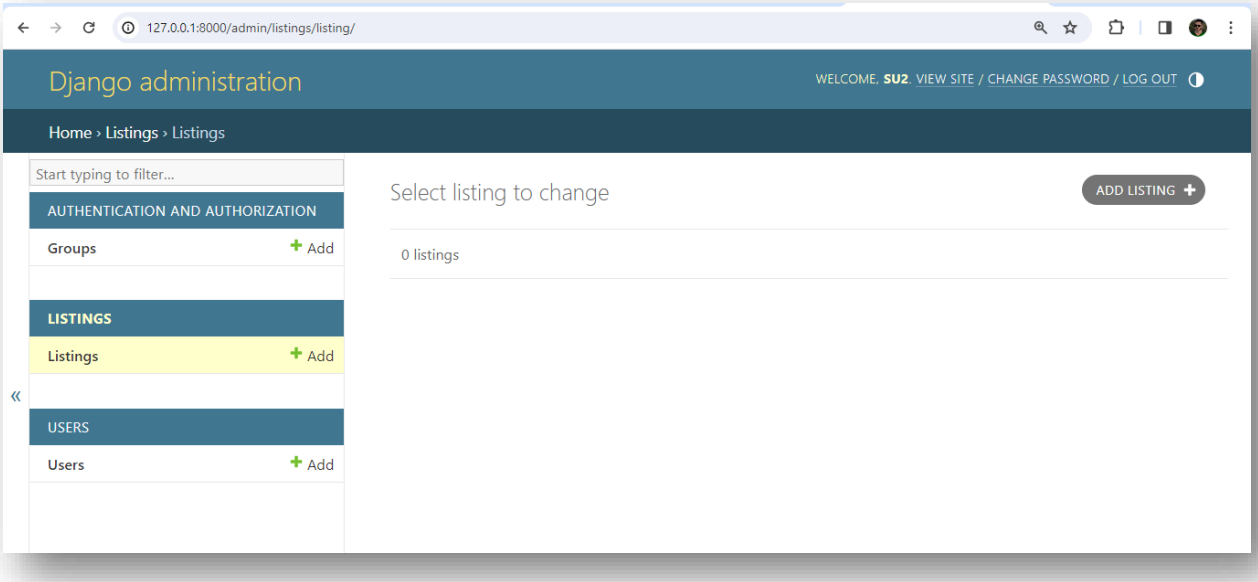
**listings/admin.py**

```python
from django.contrib import admin
from .models import Listing
# Now, we register the 'Listing' model with the Django admin site.
# This enables us to manage and view instances of the 'Listing' model through the
Django admin interface.
admin.site.register(Listing)
```

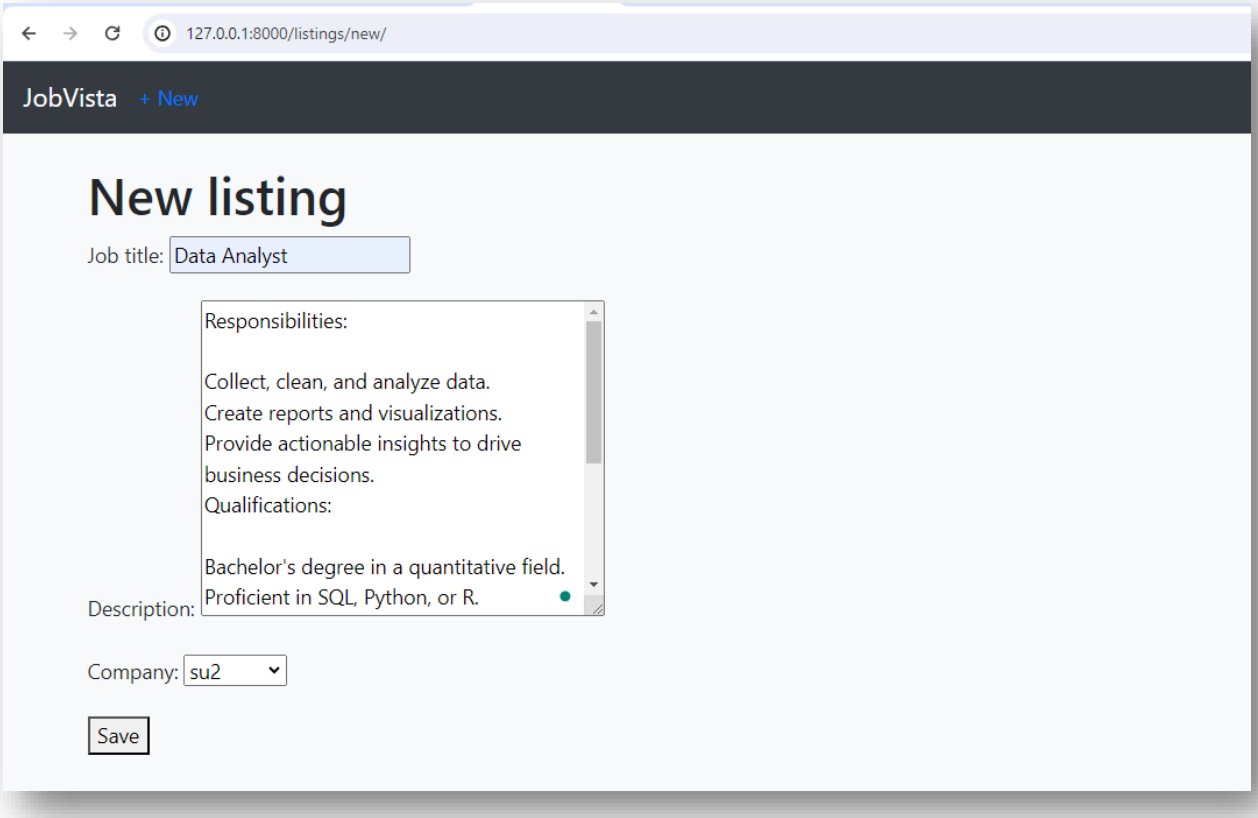
"Now, let's explore the appearance of our admin panel."



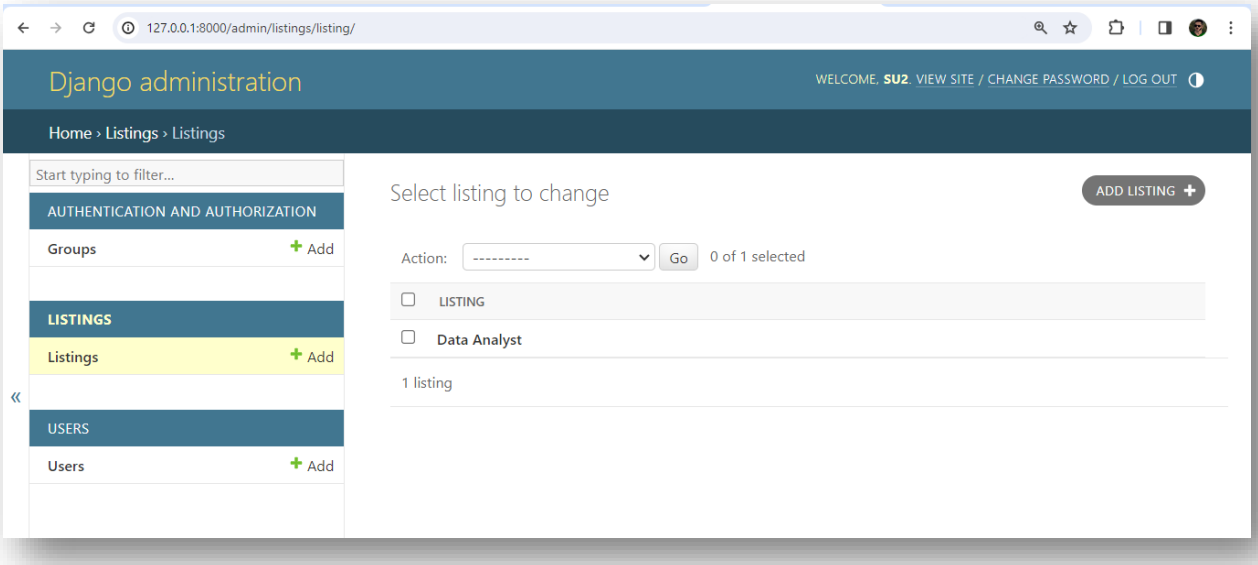

Next, click on the 'Listings' tab.



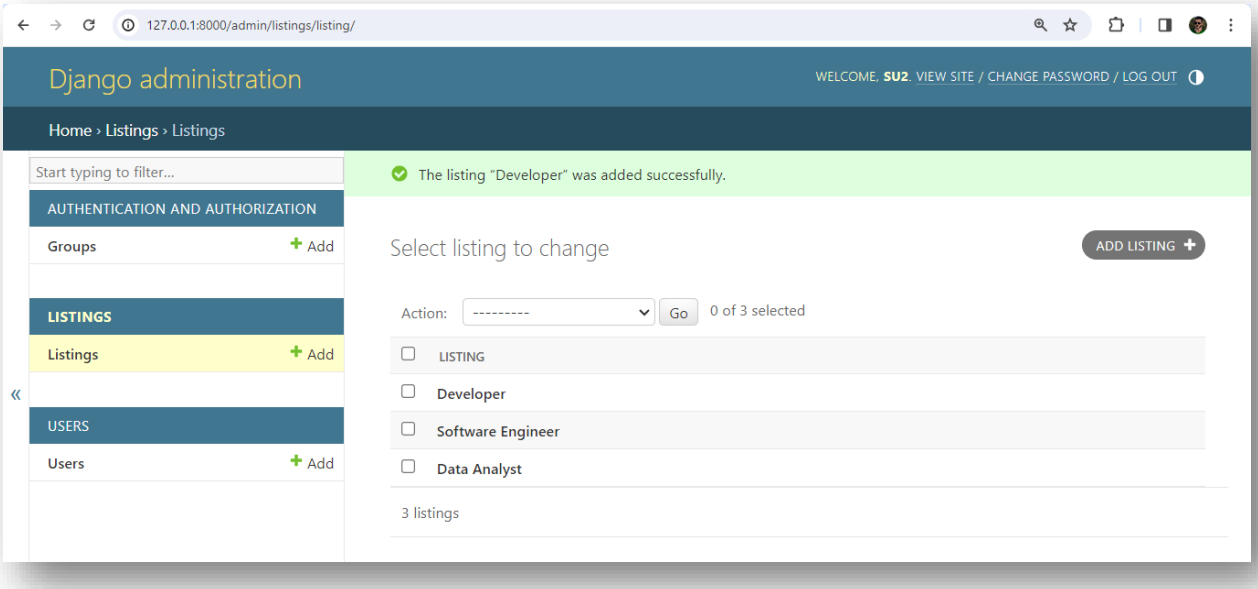
Select the "Add Listing" button to generate a new listing and then save it.




Now, take a look at the title of the listing.




Now, feel free to include a few additional listings.




Now, let's proceed to update JobVista/urls.py and incorporate the listings URLs into it.

**JobVista/urls.py**

```
from django.contrib import admin
from django.urls import path, include
from django.views.generic.base import TemplateView


urlpatterns = [
path('admin/', admin.site.urls),
path('users/', include('users.urls')),
path('users/', include('django.contrib.auth.urls')),
path('listings/', include('listings.urls')),
path('', include('pages.urls')),

]
```

This code incorporates a path 'listings/' that directs to the URL patterns defined in the 'listings.urls' module, in addition to the paths that already exist. Now, let's proceed to create the URLs for the listings app.

***command prompt***

```
> echo> listings/urls.py
```

Let’s update listings/urls.py

**listings/urls.py**

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

urlpatterns = [
path('', ListingListView.as_view(), name='listing_list'),
]
```

In this configuration, the base URL ('') is linked to a class-based view called 'ListingListView'. This view is responsible for managing the logic associated with displaying a list of objects. The distinctive identifier for this URL pattern is 'listing_list'.

Now, let's make the necessary changes in the listings/views.py file.

**listings/views.py**

```
from django.views.generic import ListView

from .models import Listing

class ListingListView(ListView):
    model = Listing
    template_name = 'listing_list.html'
```

This code establishes a Django view, namely `ListingListView`, designed to automatically fetch all instances of the Listing model and display them through the 'listing_list.html' template. Now, let's proceed to create the HTML file for it.

***command prompt***

```
> echo> templates/listing_list.html
```

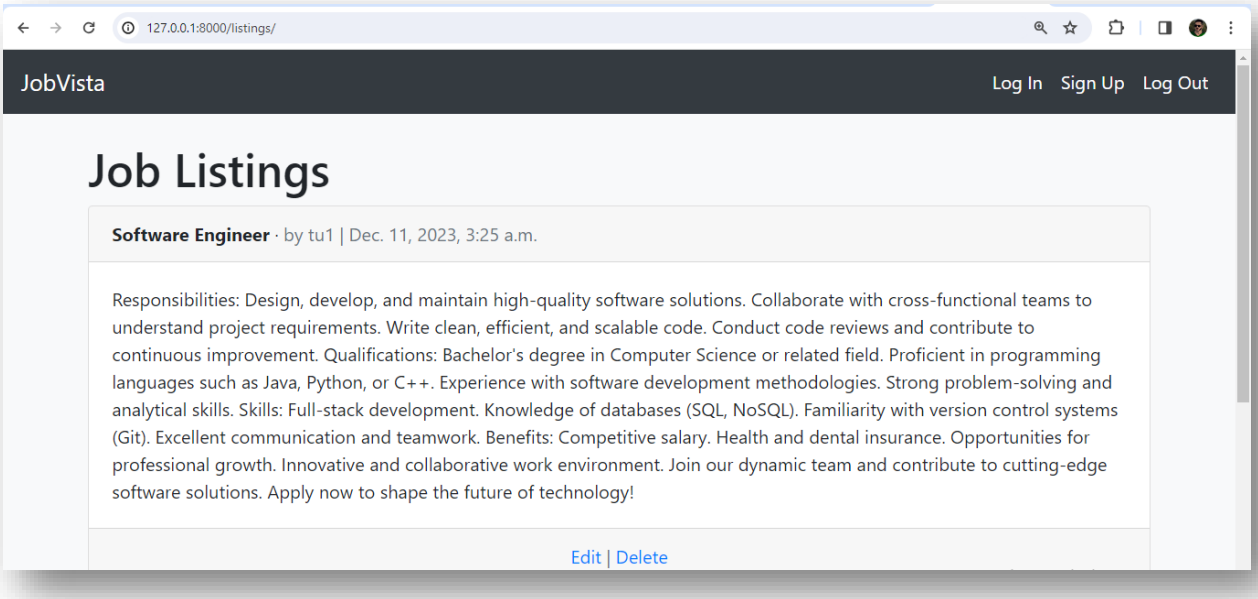
Now, let’s update the listing_list.html

**templates/listing_list.html**

```
{% extends 'base.html' %}

{% block title %}Job Listings{% endblock title %}
{% block content %}

<h1>Job Listings</h1>  <!-- Just a regular HTML heading to display 'Job Listings'
on the page. -->
```

```
{% for listing in object_list %}  <!-- This is a Django template tag for a loop. It iterates over each 'listing' in the 'object_list'. The 'object_list' is passed from the Django view and contains the job listings to be displayed. -->

<div class="card">

  <div class="card-header">
    <span class="font-weight-bold">{{ listing.job_title }}</span> &middot;
    <span class="text-muted">by {{ listing.company }} | {{ listing.date }}</span>
  </div>  <!-- This part creates a header section within the card, displaying job title, company, and date. The '{{ }}' syntax is used for outputting variables from the Django context. -->

  <div class="card-body">
    {{ listing.description }}  <!-- Here, we're outputting the job description within the card body. -->
  </div>

  <div class="card-footer text-center text-muted">
    <a href="#">Edit</a> | <a href="#">Delete</a>
  </div>  <!-- This is a card footer with links for editing and deleting the job listing. The '#' in the 'href' attribute means these links don't currently lead anywhere. -->

</div>
<br />

{% endfor %}  <!-- Closing the loop. -->

{% endblock content %}
```

Now, visit: http://127.0.0.1:8000/listings/




Currently, our functionality is limited to displaying listings. Now, let's enhance our capabilities by incorporating actions with the edit and delete buttons. Next, we'll make modifications to the listings/urls.py file.

**listings/urls.py**

```
# Import necessary modules and classes from Django
from django.urls import path
from .views import (
    ListingListView,
    ListingUpdateView,
    ListingDetailView,
    ListingDeleteView,
)

# Define the URL patterns for the app
urlpatterns = [
    # Path for updating a listing, with a dynamic part (<int:pk>) representing
the primary key of the listing
    path('<int:pk>/edit/', ListingUpdateView.as_view(), name='Listing_edit'),

    # Path for viewing the details of a listing, also with a dynamic primary key
part
    path('<int:pk>/', ListingDetailView.as_view(), name='Listing_detail'),

    # Path for deleting a listing, again with a dynamic primary key part
    path('<int:pk>/delete/', ListingDeleteView.as_view(), name='Listing_delete'),
```

```
    # Default path for listing all items, an empty string means the base URL
    path('', ListingListView.as_view(), name='Listing_list'),
]
```

This code establishes the URL patterns for a Django application focused on managing listings. It employs class-based views to handle tasks such as editing, viewing details, deleting, and listing all items. The <int:pk>/ segment within the paths signifies that these views anticipate an integer parameter named 'pk' in the URL. Typically, 'pk' represents "primary key" and is a standard identifier for database records.

Now, let's proceed to update listings/views.py and define the essential classes.

**listings/views.py**

```
# Import necessary functions and classes from Django
from django.shortcuts import render
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView
from django.urls import reverse_lazy
from .models import Listing  # Import the Listing model from the current
directory (assuming a models.py file exists)

# Define a class for a view that displays a list of all Listing objects
class ListingListView(ListView):
    model = Listing  # Set the model for this view to the Listing model
    template_name = 'listing_list.html'  # Specify the template (HTML file) to be
used for rendering this view

# Define a class for a view that displays the details of a specific Listing
object
class ListingDetailView(DetailView):
    model = Listing  # Set the model for this view to the Listing model
    template_name = 'listing_detail.html'  # Specify the template (HTML file) to
be used for rendering this view

# Define a class for a view that allows editing an existing Listing object
class ListingUpdateView(UpdateView):
    model = Listing  # Set the model for this view to the Listing model
    fields = ('job_title', 'description',)  # Specify the fields of the Listing
model that can be edited
    template_name = 'listing_edit.html'  # Specify the template (HTML file) to be
used for rendering this view

# Define a class for a view that allows deleting an existing Listing object
class ListingDeleteView(DeleteView):
```

```
    model = Listing  # Set the model for this view to the Listing model
    template_name = 'Listing_delete.html'  # Specify the template (HTML file) to
be used for rendering this view
    success_url = reverse_lazy('listing_list')  # Specify the URL to redirect to
after successful deletion
```

These classes define distinct perspectives for managing the Listing model within a Django web application. They encompass tasks such as listing items, viewing details, updating information, and deleting items. The `template_name` attribute within each class indicates the designated HTML template for rendering the corresponding views. Next, generate HTML files to correspond with these views.

***command prompt***

```
> echo> templates/listing_detail.html
> echo> templates/listing_edit.html
> echo> templates/listing_delete.html
```

Let's update listing_detail.html

***templates/listing_detail.html***

```
{% extends 'base.html' %}
{% block content %}
<div class="listing-entry">
    <h2>{{ object.job_title }}</h2>  <!-- Displaying the job title using Django
template variable syntax. 'object' is a variable passed to this template. -->

    <p>by {{ object.company }} | {{ object.date }}</p>  <!-- Displaying company
and date information using the same Django template variable syntax. -->

    <p>{{ object.description }}</p>  <!-- Displaying the job description. -->
</div>

<p><a href="{% url 'listing_edit' listing.pk %}">Edit</a> |  <!-- Creating a link
to the 'listing_edit' URL with the primary key (pk) of the listing. The URL is
generated using the 'url' template tag. -->
<a href="{% url 'listing_delete' listing.pk %}">Delete</a></p>  <!-- Creating a
link to the 'listing_delete' URL with the primary key (pk) of the listing. The
URL is generated using the 'url' template tag. -->

<p>Back to <a href="{% url 'listing_list' %}">All Listings</a>.</p>  <!--
Creating a link to the 'listing_list' URL, likely leading back to a page
displaying all listings. The URL is generated using the 'url' template tag. -->
```

```
{% endblock content %}
```

This template has been crafted to showcase details pertaining to a job listing, encompassing essential information like job title, company, date, and a comprehensive description. Additionally, it facilitates user interaction by offering links for editing and deleting the listing, along with a convenient link to navigate back to the page displaying all listings. Notably, the URLs for actions like editing, deleting, and accessing the listing list are dynamically generated through the utilization of Django's 'url' template tag.

Let's update listing_edit.html

***templates/listing_edit.html***

```
{% extends 'base.html' %}
{% block content %}
<h1>Edit</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<button  type="submit">Update</button>
</form>
{% endblock content %}
```

This Django template is extending a base template, defining a content block, and displaying a form for editing data. The form uses the POST method for data submission, includes a CSRF token for security, and has a button to trigger the update process. The actual form fields are rendered using the form.as_p template tag.

Let's update listing_delete.html

***templates/listing_delete.html***

```
{% extends 'base.html' %}
{% block content %}
<h1>Delete</h1>
<form action="" method="post">{% csrf_token %}
<p>Are you sure you want to delete "{{ listing.job_title }}"?</p>
<button type="submit">Confirm</button>
</form>
{% endblock content %}
```

This template is creating a confirmation page for deleting an item. It extends a base template, defines a content block, and includes a form with a confirmation message and a submit button. The form submission is secured against CSRF attacks using Django's built-in {% csrf_token %} tag.

Let’s update the listing_list.html

***templates/listing_list.html***

```
…
  <div class="card-footer text-center text-muted">
    <a href="{% url 'listing_edit' listing.pk %}">Edit</a> |
    <a href="{% url 'listing_delete' listing.pk %}">Delete</a>
  </div>
…
```

So, this piece of code creates a card footer in HTML that contains "Edit" and "Delete" links. These links are dynamic, meaning they are generated based on the specific listing's primary key using Django template tags. The "Edit" link leads to the 'listing_edit' view, and the "Delete" link leads to the 'listing_delete' view, allowing users to perform these actions on a listing in a Django web application.

After completing these steps, if the server is already running, stop it using Control+C. Then, enter the command "python manage.py runserver" in the command line:

***command prompt***

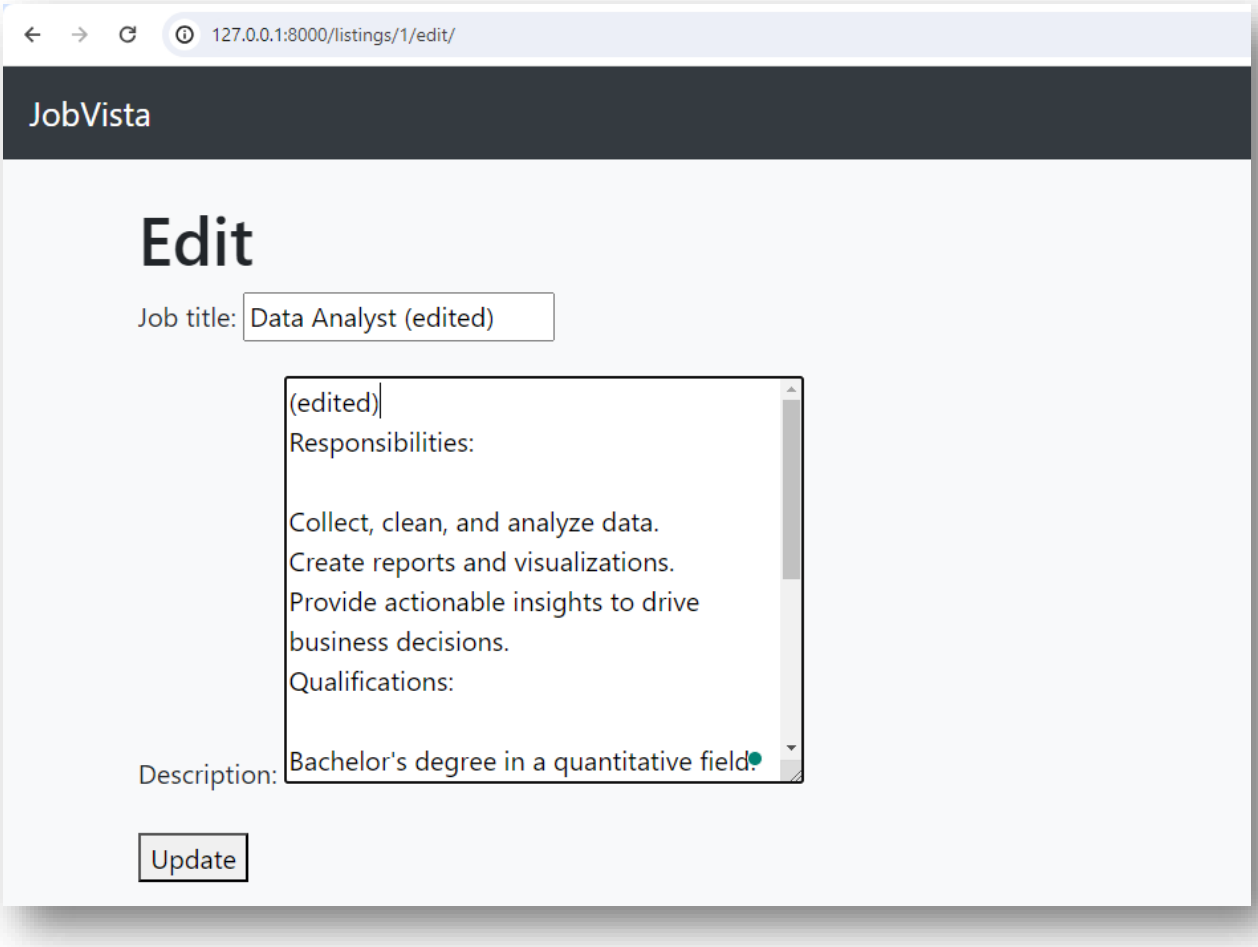
```
> python manage.py runserver
```

Visit: http://127.0.0.1:8000/listings/.

Select the "edit" link within the first listing, and you will be directed to:

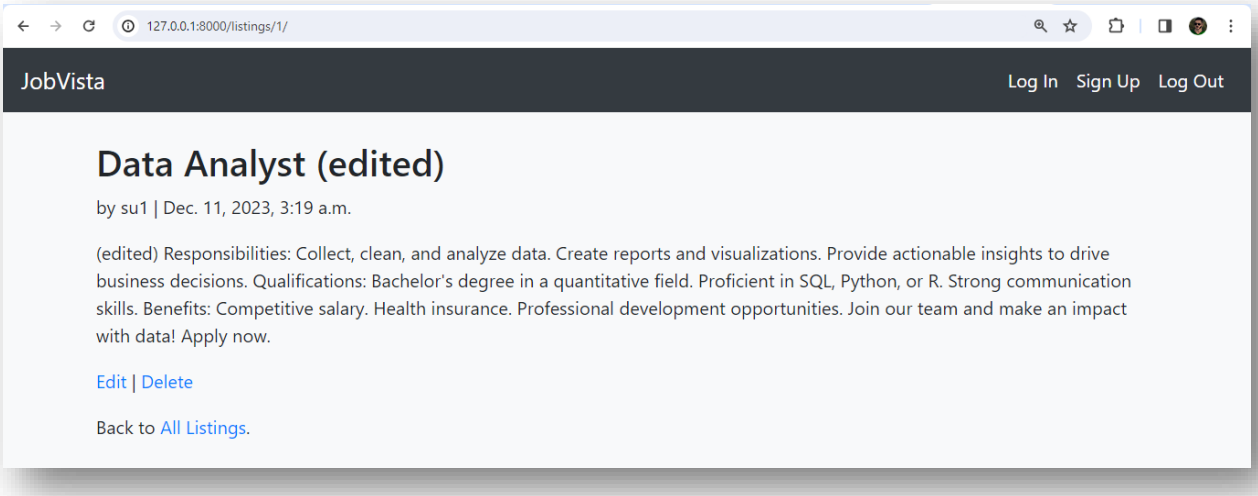
http://127.0.0.1:8000/listings/1/edit/

Now, edit the listing.

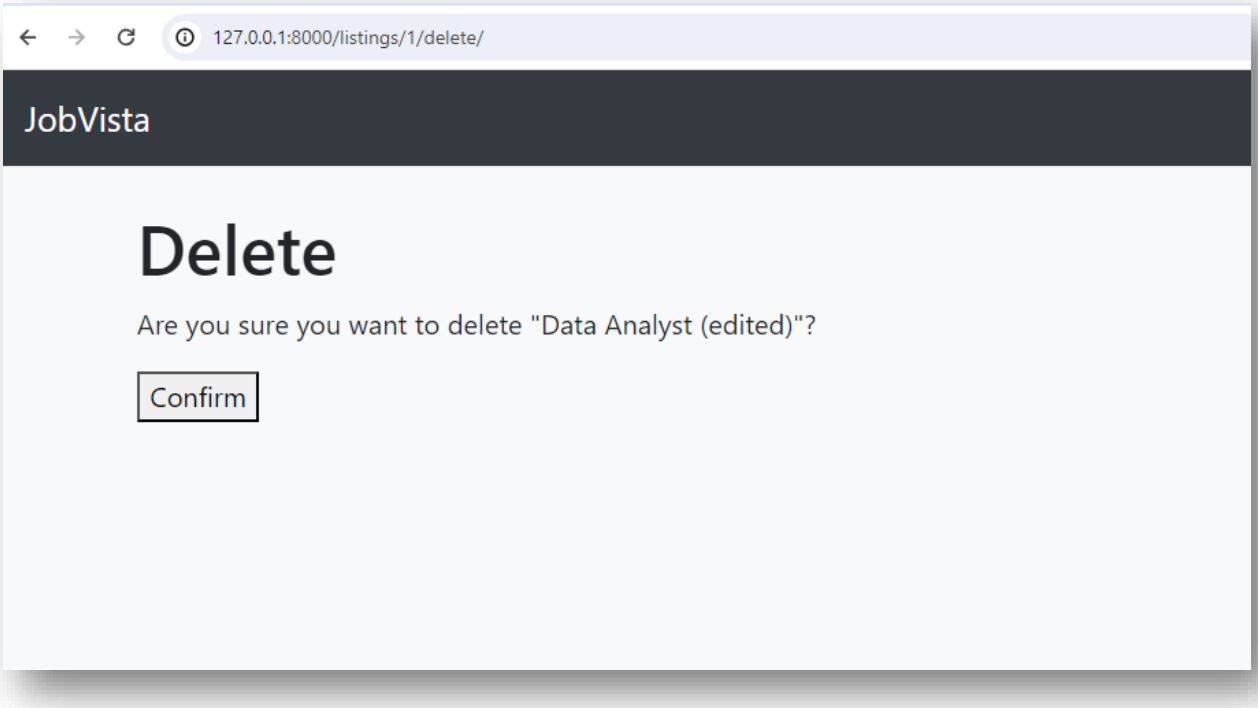



Select "Update," and you'll observe the modifications reflected in the listings.

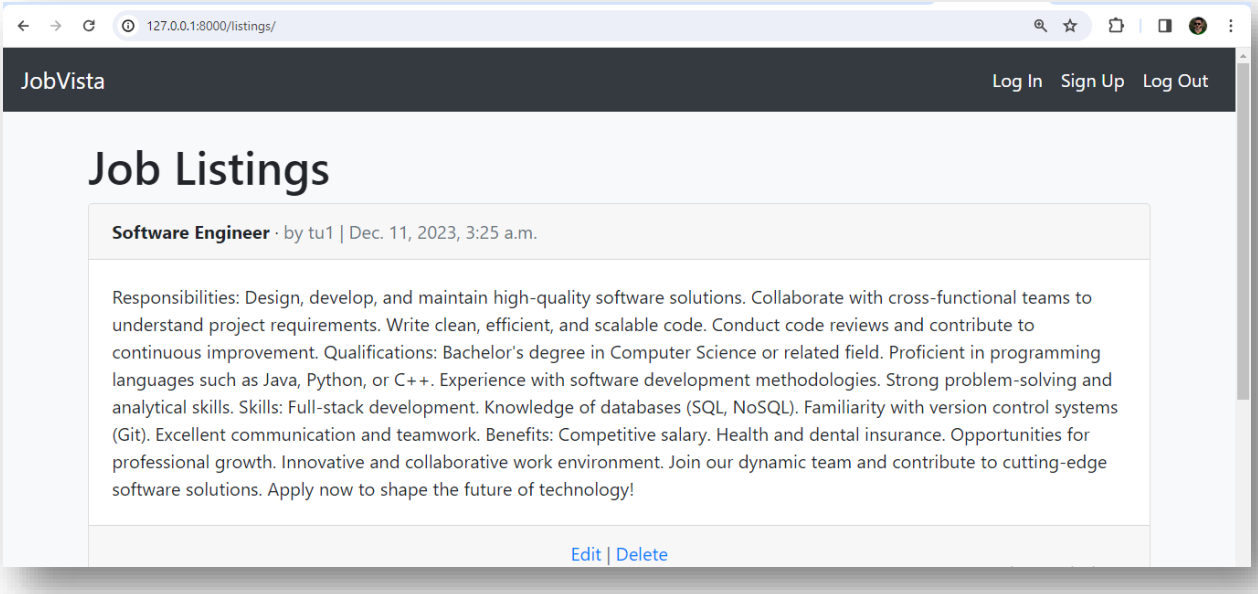
http://127.0.0.1:8000/listings/1/



Now, let's see how we can delete a listing. Simply click on the Delete button.




Now click on confirm.



Now you can observe that the listing has been successfully deleted. Next, let's implement a new functionality to create listings directly from the front end. This will eliminate the need to create listings through the admin panel.

Let’s update listings/views.py

**listings/views.py**

```
…
# Importing necessary modules from Django for handling generic views and form
submissions. The new view here is CreateView.

from django.views.generic.edit import UpdateView, DeleteView , CreateView
…
# Defining a custom view for creating a new listing using Django's CreateView
class ListingCreateView(CreateView):
    # Specifying the model that this view will interact with (assuming there's a
Listing model defined elsewhere)
    model = Listing

    # Specifying the template (HTML file) that will be used to render this view
    template_name = 'listing_new.html'

    # Specifying the fields from the Listing model that should be included in the
form for creating a new listing
    fields = ('job_title', 'description', 'company',)

…
```

Let’s update listings/urls.py

**listings/urls.py**

```
from django.urls import path

from .views import (
ListingListView,
ListingUpdateView,
ListingDetailView,
ListingDeleteView,
# We're importing the 'ListingCreateView' class from the 'views' module within
the current package (indicated by the dot).
ListingCreateView,
)
```

```
urlpatterns = [
path('<int:pk>/edit/',ListingUpdateView.as_view(), name='listing_edit'),
path('<int:pk>/',ListingDetailView.as_view(), name='listing_detail'),
path('<int:pk>/delete/',ListingDeleteView.as_view(), name='listing_delete'),
    # The first argument is the path that users will access in the browser. In
this case, it's 'new/'.
    # The second argument is the view that will handle the logic when this URL is
accessed.
    # We're using the 'ListingCreateView.as_view()' method to convert our class-
based view into a view callable by Django.


path('new/', ListingCreateView.as_view(), name='listing_new'),
path('', ListingListView.as_view(), name='listing_list'),
]
```

Let's exit the server by using the command "control + c". Next, we'll proceed to create the HTML file to implement this functionality.

***command prompt***

```
> echo>  templates/listing_new.html
```

Let's update the **listing_new.html**

**templates/listing_new.html**

```
{% extends 'base.html' %}
{% block content %}
<h1>New listing</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<button type="submit">Save</button>
</form>
{% endblock content %}
```

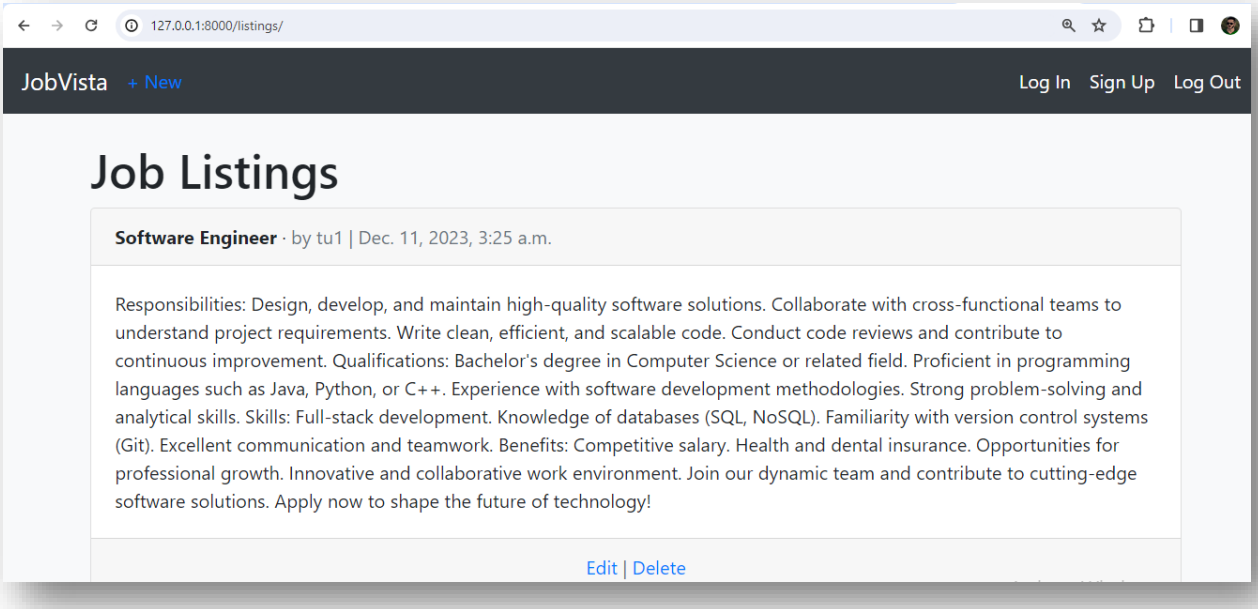
This template extends a base template, includes a form for creating a new listing, and handles the submission of the form. The CSRF token is added for security, and the form fields are displayed in a paragraph format. The 'Save' button triggers the form submission.

Let's update the base.html

***templates/base.html***

```
…

<!-- The <nav> element represents the navigation bar, and it has classes for
styling provided by Bootstrap. -->
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <!-- The <a> element represents a hyperlink with the text "JobVista," and
it links to the 'home' URL defined in the Django project. -->
        <a class="navbar-brand" href="{% url 'home' %}">JobVista</a>

        <!-- The following block of code is a Django template tag that checks if
the user is authenticated. -->
        {% if user.is_authenticated %}
            <!-- If the user is authenticated, a list with the class "navbar-nav"
is created for navigation links. -->
            <ul class="navbar-nav mr-auto">
                <!-- Within the list, there is a list item (<li>) containing a
hyperlink that links to the 'listing_new' URL. -->
                <li class="nav-item"><a href="{% url 'listing_new' %}">+
New</a></li>
            </ul>
        {% endif %}

        <!-- The following button is a toggler for collapsing/expanding the
navigation bar on smaller screens. -->
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
            <!-- The span with the class "navbar-toggler-icon" represents the
icon displayed on the toggler button. -->
            <span class="navbar-toggler-icon"></span>
        </button>
    </nav>


…
```

Now let's visit: http://127.0.0.1:8000/listings/



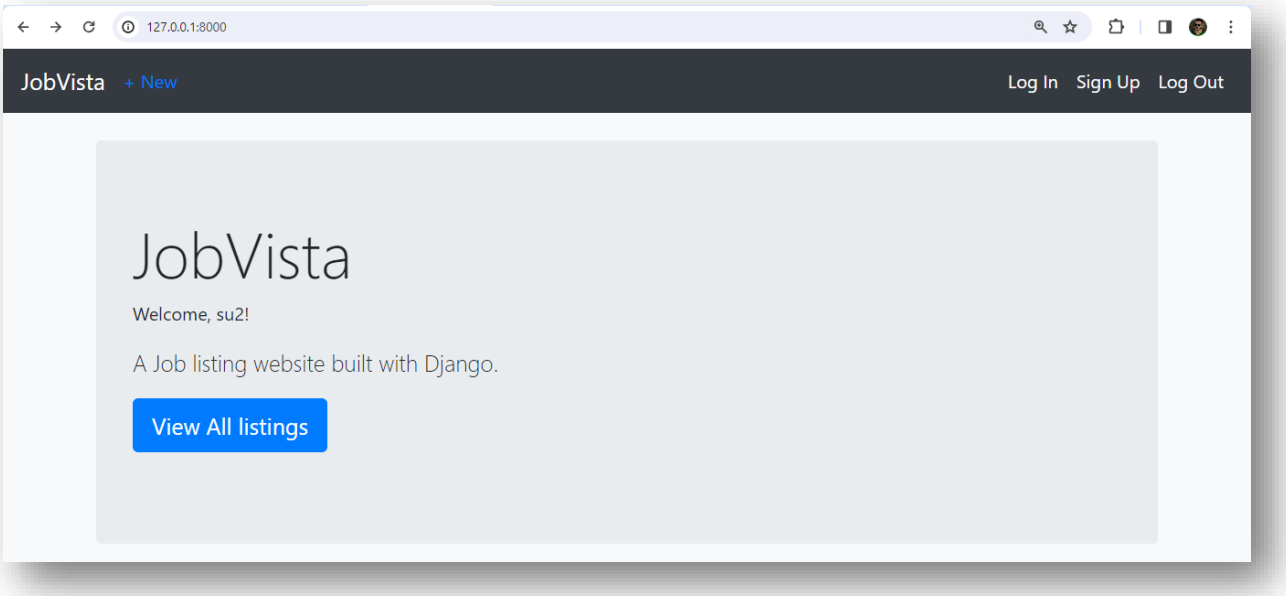

Now you can locate the +New button right beside the website title. Let's proceed to update home.html.

***templates/home.html***

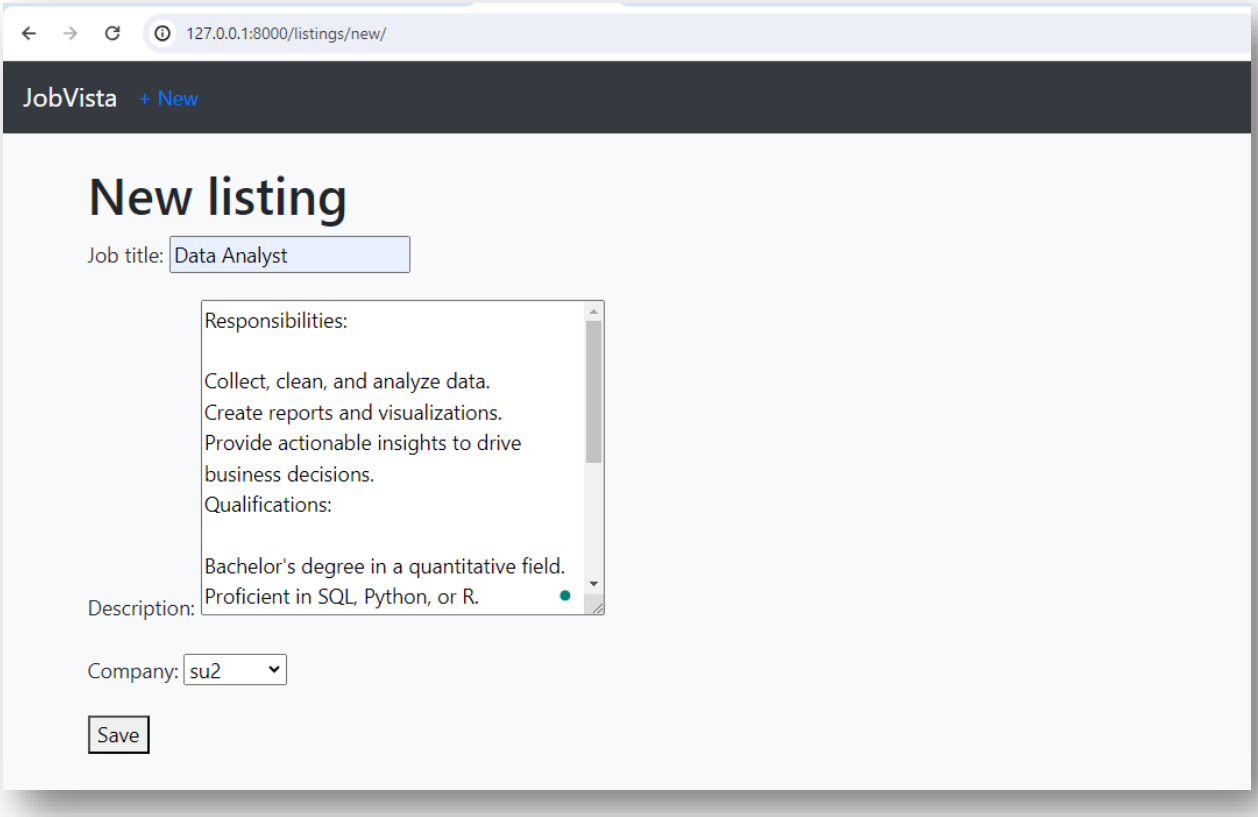
```
{% extends 'base.html' %}

{% block title %}
  Home
{% endblock title %}

{% block content %}
<div class="jumbotron">
  <h1 class="display-4">JobVista</h1>

  {% if user.is_authenticated %}
    <!-- This is a Django template tag that checks if the user is authenticated (logged in). -->
    <p>Welcome, {{ user.username }}!</p>
    <!-- If the user is logged in, it displays a personalized welcome message with the username. -->

  {% else %}
    <p>You are not logged in.</p>
    <!-- If the user is not logged in, it displays a message indicating that the user is not logged in. -->

  {% endif %}
```

```
  <p class="lead">A Job listing website built with Django.</p>
  <!-- This is a lead paragraph providing a brief description of the website. -->

  <p class="lead">
    <a class="btn btn-primary btn-lg" href="{% url 'listing_list' %}"
role="button">
      View All listings
    </a>
  </p>
  <!-- This is a button that links to the 'listing_list' URL, which is defined in
the Django project. It allows users to view all job listings. -->
</div>
{% endblock content %}
```

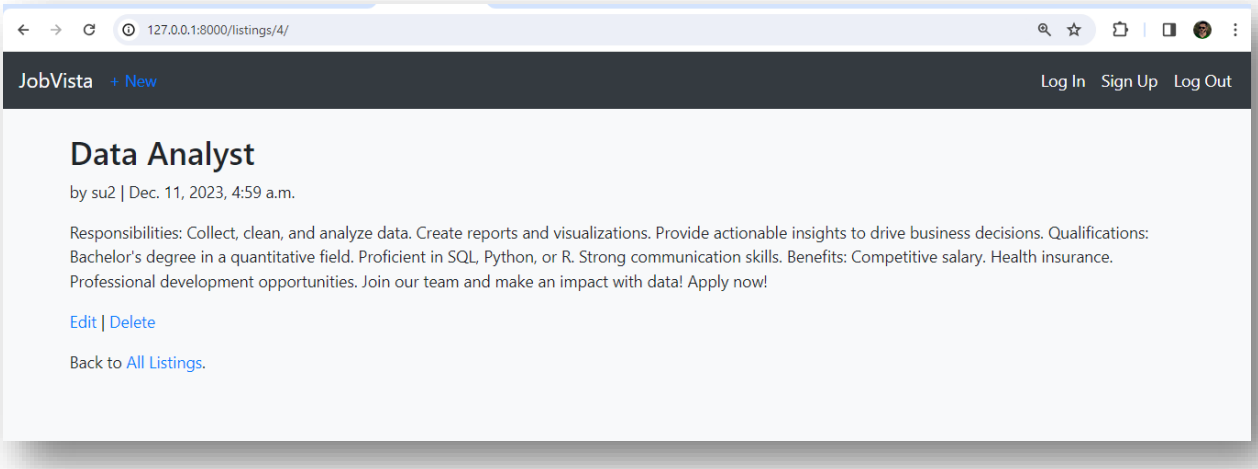
Now let's visit: http:/127.0.0.1:8000/
This is how it looks:



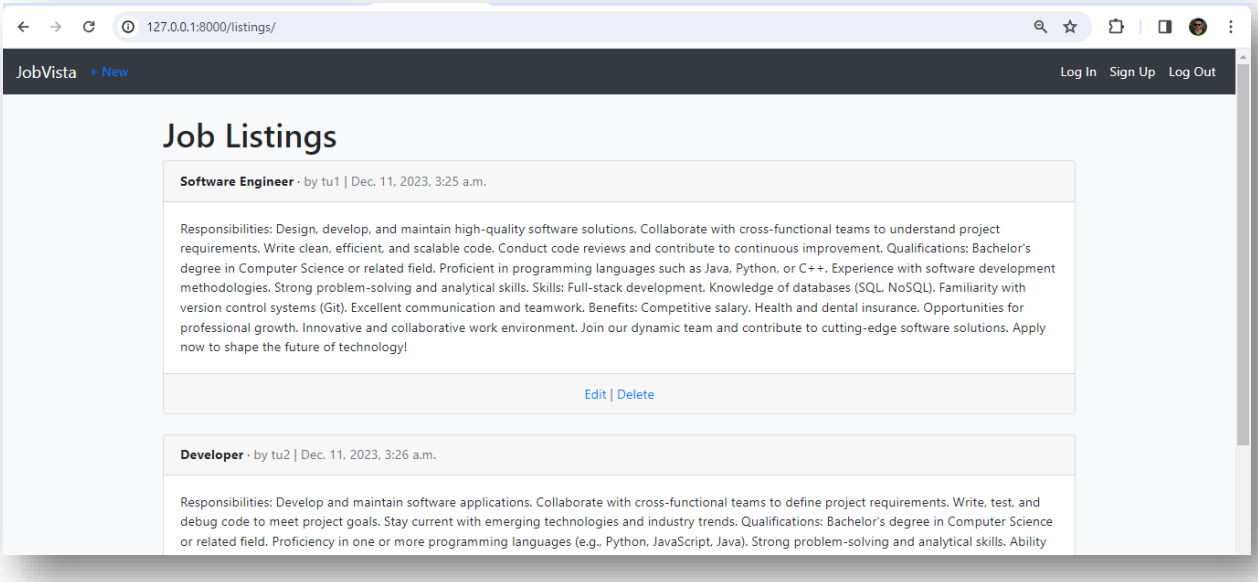

Now let's click on **+New** button and create a new listing.




Click on the save button.




Please view the recently created listing, then click on the "All Listings" link.




All the listings on this page are visible to you.

## Conclusion

In conclusion, this chapter has outlined the development of the Job Listing web application, emphasizing its core functionalities of Create, Update, Read, and Delete (CRUD). The approach ensured a seamless user experience, allowing users to post, update, and browse job listings effortlessly. This chapter highlights the importance of adaptability in web development and showcases the Job Listing web app as a user-friendly tool connecting job seekers with opportunities.

# Chapter 14: Permissions and Authorization

In this chapter, we aim to enhance the security of our website.

Currently, when creating a new listing, users have the option to manually input the company name, posing a potential security vulnerability. This feature lacks robust security measures.

To address this concern, our solution is to automatically set the company name by default for logged-in users. This enhancement will contribute to a more secure and controlled environment for our platform.

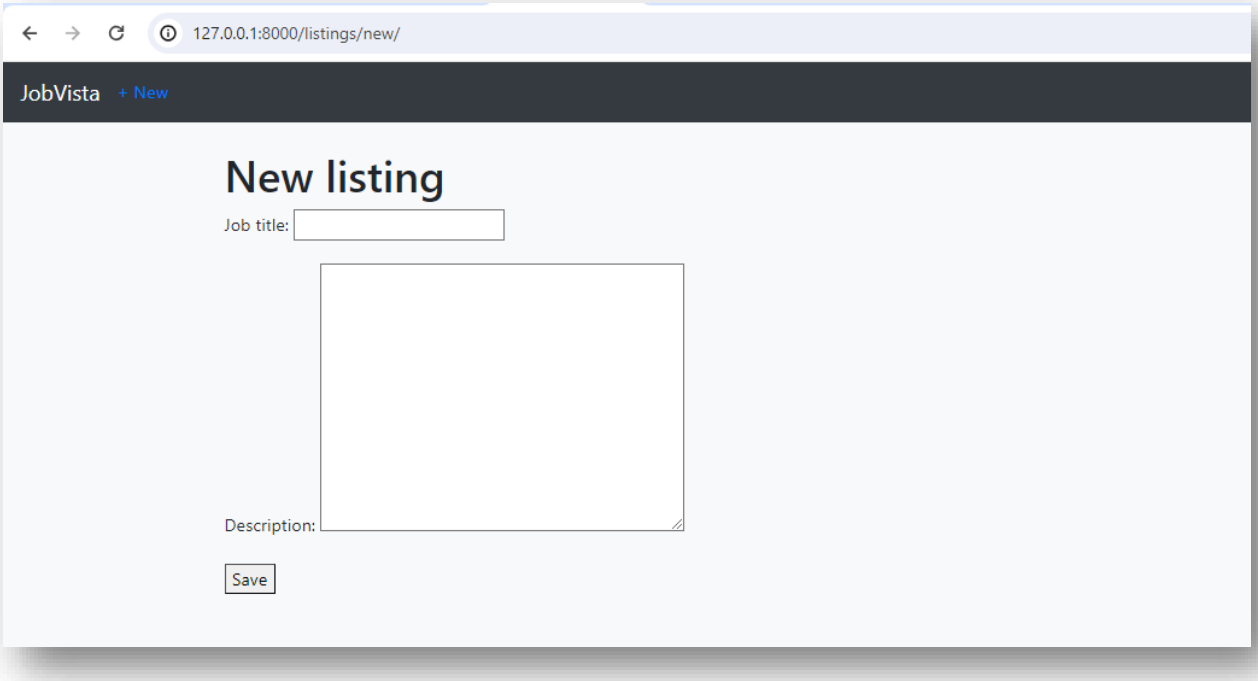
***listings/views.py***

```
class ListingCreateView(CreateView):
    model = Listing
    template_name = 'listing_new.html'
    fields = ('job_title', 'description')


# The following function is used when a form is successfully submitted, and it's
called when the form data is valid.

    def form_valid(self, form):
        form.instance.company = self.request.user
        return super().form_valid(form)
```

This code guarantees that the form data is linked to the presently logged-in user. Subsequently, it saves the form data by utilizing the default functionality offered by Django's built-in form handling mechanisms.

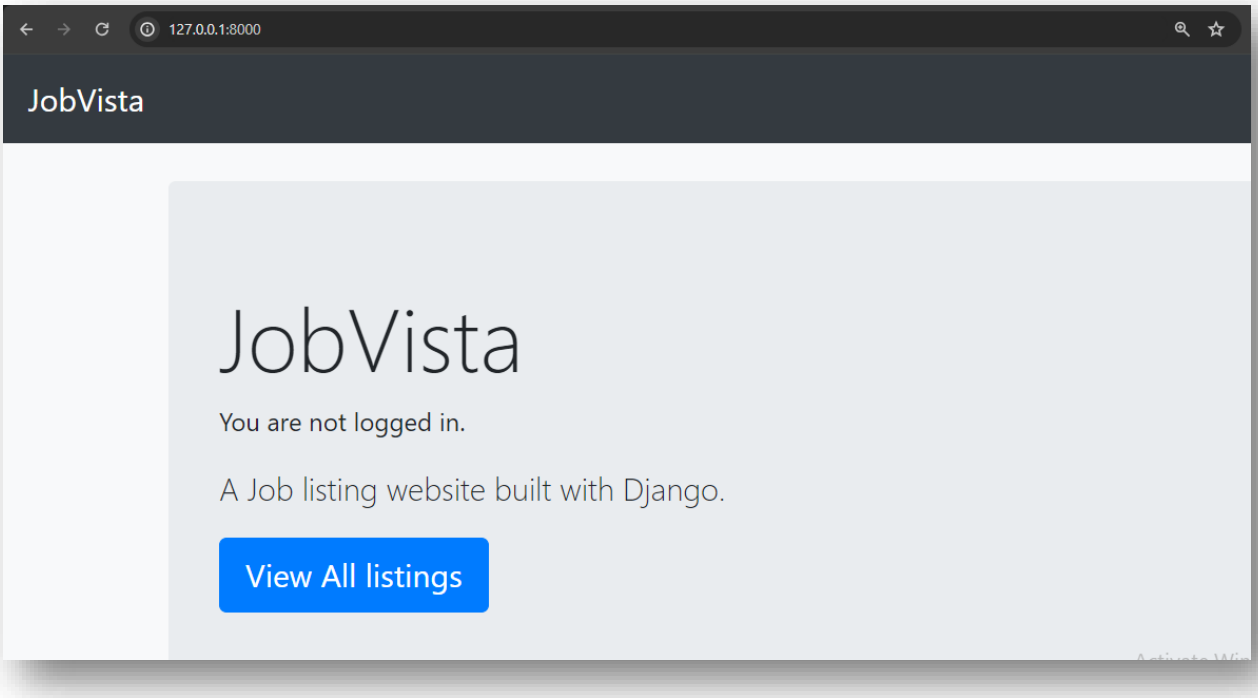
To experience this functionality, navigate to: http://127.0.0.1:8000/

Click on the "+New" button. You will now observe only two options: "Job Title" and "Description." When creating a new listing, the company name is automatically assigned for the logged-in user.

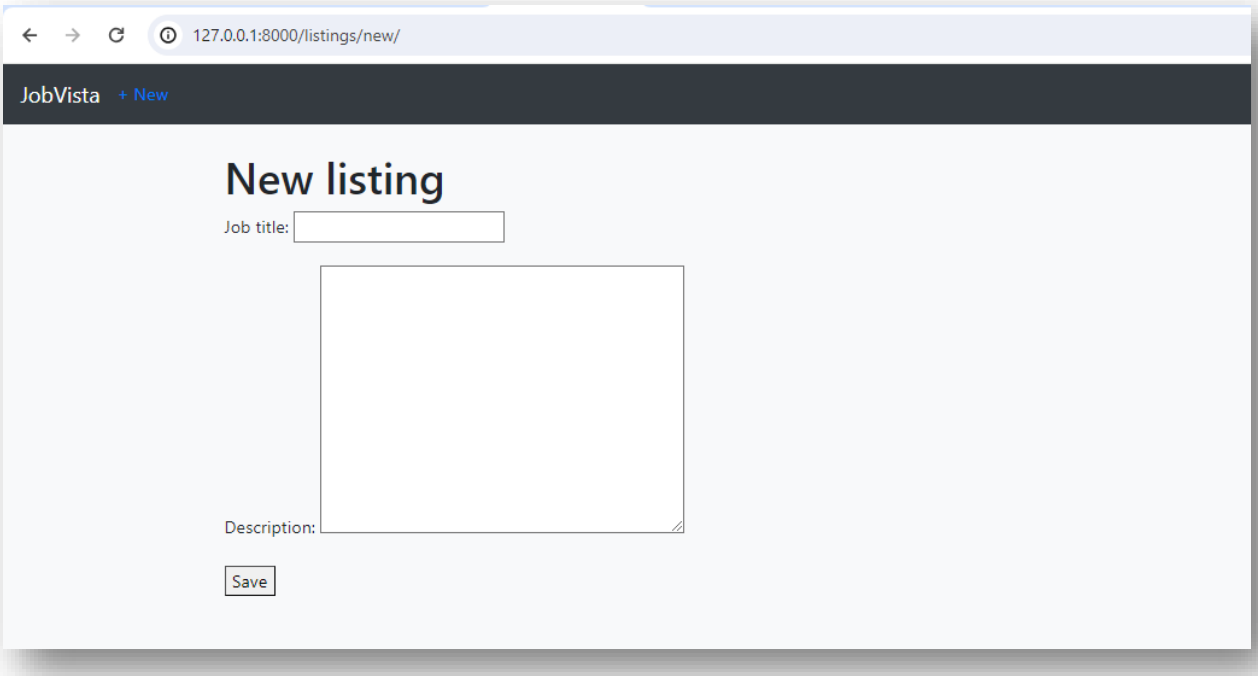



## Authorizations

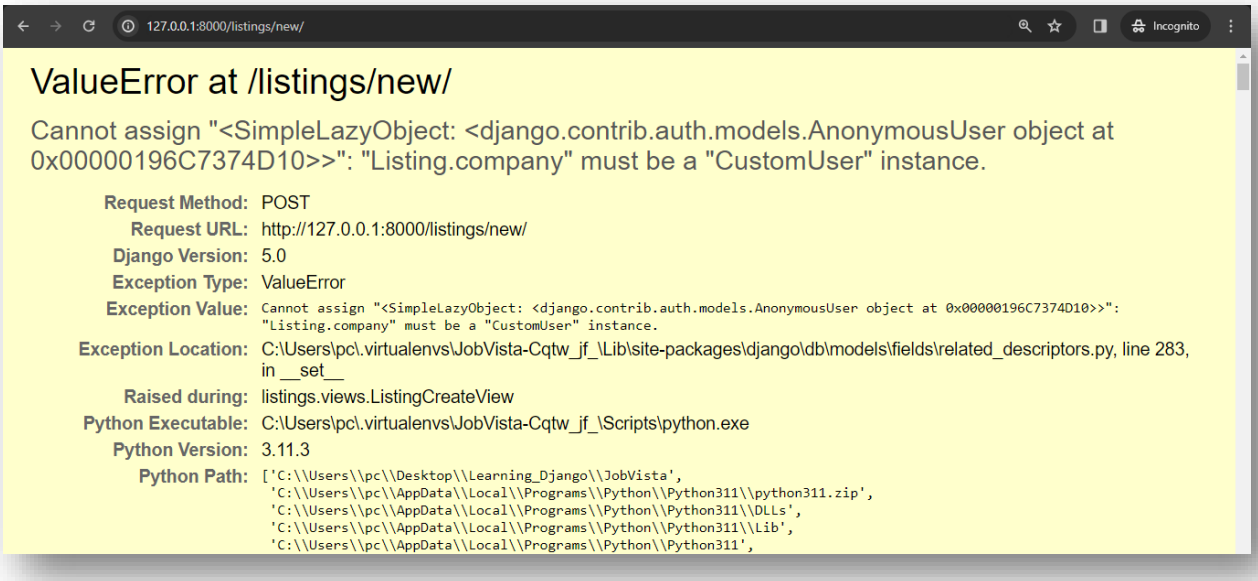
Consider what would happen if a logged-out user tried to create a new listing? To try it out, click on your username in the upper right corner of the nav bar, then select “Log out” from the dropdown options.




The “+ New” link disappears from the nav bar but what happens if you go to it directly: http://127.0.0.1:8000/listings/new/?



Fill the details and click on save.




You will see a big error message.

## Mixins

We aim to establish specific authorizations, ensuring that only users who are logged in can access the site. To achieve this, we employ a mixin, a special form of multiple inheritance utilized by Django to prevent code duplication while still allowing customization. For instance, the generic ListView and DetailView both require a method to return a template. Instead of duplicating this code in each large generic view, Django utilizes a "mixin" called TemplateResponseMixin to consolidate this functionality. ListView and DetailView both incorporate this mixin to render the appropriate template.

As evident in the freely available Django source code on Github, mixins are widely employed throughout.

To restrict access to views exclusively to logged-in users, Django provides a LoginRequired mixin, which is both potent and succinct. In the "articles/views.py" file, import this mixin at the beginning, and then apply LoginRequiredMixin to our ArticleCreateView. It is crucial to ensure that the mixin is positioned to the left of ListView so that it is processed first. This arrangement ensures that ListView is already aware of our intent to restrict access.

With that, our task is now finished. Now, let's proceed to update the listings/views.py file.

***listings/views.py***

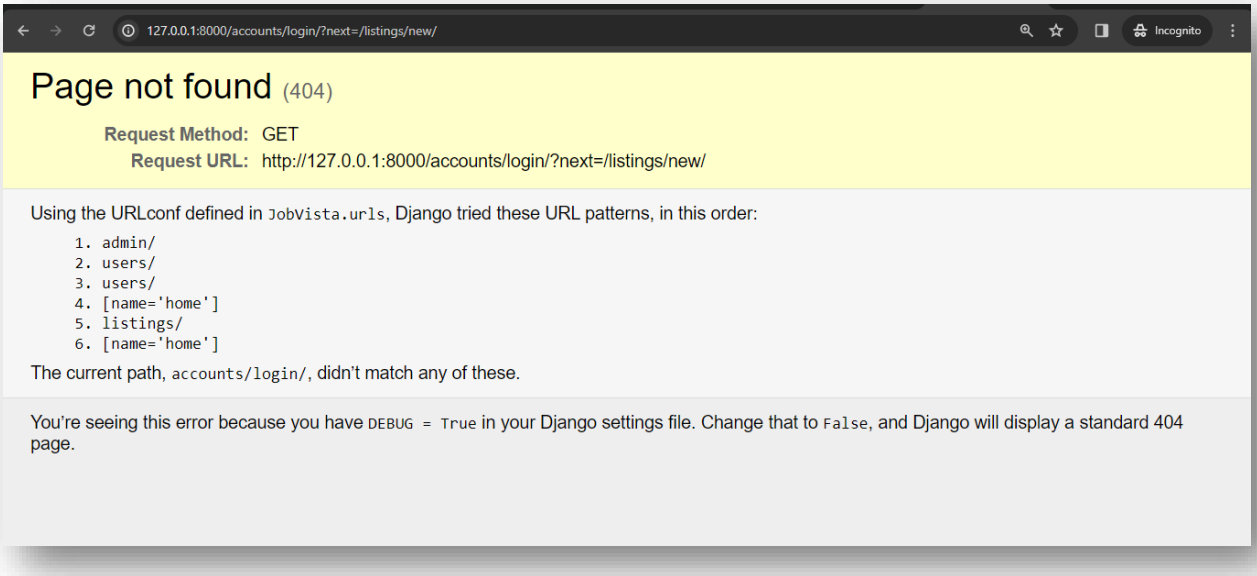
```
from django.shortcuts import render

# Import the LoginRequiredMixin from the Django authentication mixins
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView , CreateView
from django.urls import reverse_lazy
from .models import Listing


…
# Define a class for creating a new listing, inheriting from LoginRequiredMixin
and CreateView
class ListingCreateView(LoginRequiredMixin,CreateView):
    model = Listing
    template_name = 'listing_new.html'
    fields = ('job_title', 'description')


    def form_valid(self, form):
        form.instance.company = self.request.user
        return super().form_valid(form)
```

In Django, LoginRequiredMixin is a class-based view mixin that ensures that only authenticated (logged-in) users can access a particular view. In this case, it's used as a parent class for ListingCreateView, indicating that only logged-in users can create a new listing.

So, when a user tries to access the page to create a new listing (ListingCreateView), Django checks if the user is authenticated. If not, the user is redirected to the login page. This is a helpful way to protect certain views or pages, ensuring that only authorized users can perform certain actions, like creating a new listing in this example.

Now visit: http://127.0.0.1:8000/listings/new/

You will now encounter a "Page Not Found" error.




Our next objective is to automatically redirect users to the login page if they inadvertently attempt to access restricted content on the website. To achieve this, we need to make updates to our listings/views.py file.

***listings/views.py***

```
class ListingCreateView(LoginRequiredMixin,CreateView):
    model = Listing
    template_name = 'listing_new.html'
    fields = ('job_title', 'description')
    login_url = 'login'
```

```
    def form_valid(self, form):
        form.instance.company = self.request.user
        return super().form_valid(form)
```

When you see the line **login_url = 'login'** in a Django view, it means that the LoginRequiredMixin is being customized to specify a particular URL to which the user should be redirected if they are not authenticated.

The login_url attribute is used to specify the URL where Django should redirect the user if they are not authenticated and try to access a view that requires authentication. In this case, 'login' is the URL pattern name for the login page.

So, when an unauthenticated user tries to access the ListingCreateView, they will be redirected to the login page ('login'). This is a way to customize the behavior of LoginRequiredMixin and direct users to the appropriate login page for your application.

Try the link for creating new messages again: http://127.0.0.1:8000/listings/new/. It now redirects users to the log in page. Just as we desired!

## LoginRequiredMixin

Let's do it for all the views

***listings/views.py***

```
from django.shortcuts import render
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView , CreateView
from django.urls import reverse_lazy
from .models import Listing

class ListingListView(LoginRequiredMixin, ListView):
    model = Listing
    template_name = 'listing_list.html'
    login_url = 'login'

class ListingDetailView(LoginRequiredMixin, DetailView):
    model = Listing
    template_name = 'listing_detail.html'
    login_url = 'login'

class ListingUpdateView(LoginRequiredMixin, UpdateView):
    model = Listing
```

```
    fields = ('job_title', 'description',)
    template_name = 'listing_edit.html'
    login_url = 'login'

class ListingDeleteView(LoginRequiredMixin, DeleteView):

    model = Listing
    template_name = 'Listing_delete.html'
    success_url = reverse_lazy('listing_list')
    login_url = 'login'


class ListingCreateView(LoginRequiredMixin,CreateView):
    model = Listing
    template_name = 'listing_new.html'
    fields = ('job_title', 'description')
    login_url = 'login'


    def form_valid(self, form):
        form.instance.company = self.request.user
        return super().form_valid(form)
```

## UpdateView and DeleteView

Now even if you are a valid logged in user, you should be able to update or delete the listings created by other users. So, we have to create a funtionality to restrict the users from updating or deleting the listings created by other users and you should be able update and delete the listings of your own.

To accomplish that, please navigate to views.py.

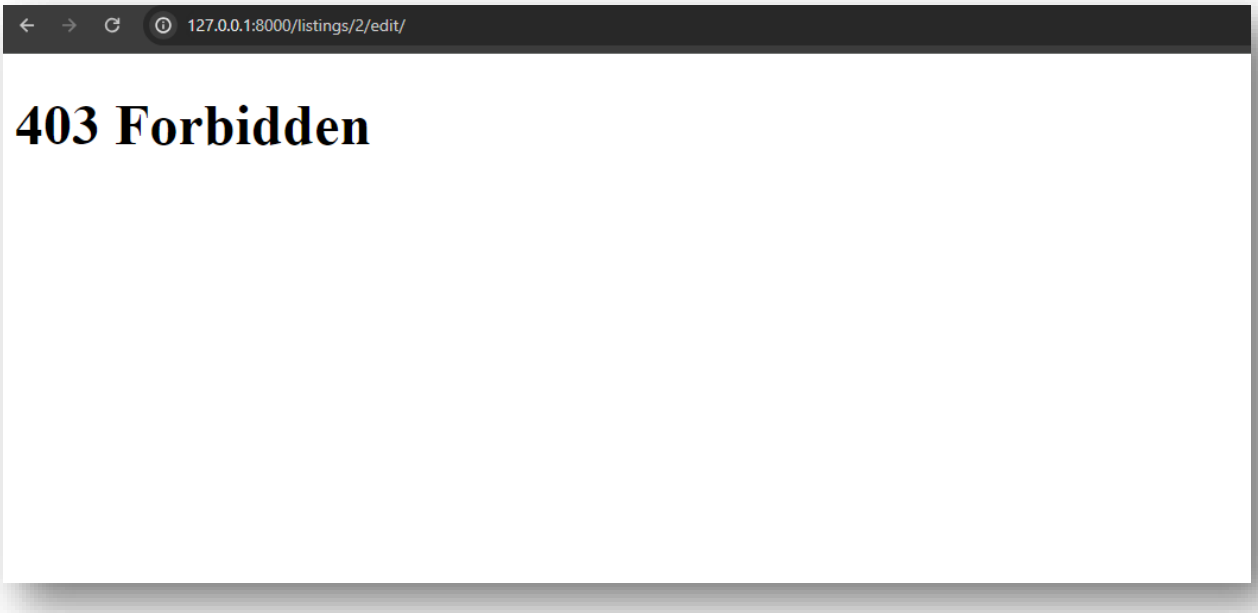
***listings/views.py***

```
from django.shortcuts import render

from django.contrib.auth.mixins import LoginRequiredMixin
# import the PremissionDenied exception class provided by Django
from django.core.exceptions import PermissionDenied
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView , CreateView
from django.urls import reverse_lazy
from .models import Listing

class ListingListView(LoginRequiredMixin, ListView):
    model = Listing
    template_name = 'listing_list.html'
```

```
    login_url = 'login'

class ListingDetailView(LoginRequiredMixin, DetailView):
    model = Listing
    template_name = 'listing_detail.html'
    login_url = 'login'

# Define a class for updating a listing, inheriting from LoginRequiredMixin and
UpdateView
class ListingUpdateView(LoginRequiredMixin, UpdateView):
    # Specify the model that this view is updating
    model = Listing
    # Specify the fields from the model that can be updated
    fields = ('job_title', 'description',)
    # Specify the template used for rendering the update form
    template_name = 'listing_edit.html'
    # Specify the URL to redirect to for login if the user is not authenticated
    login_url = 'login'

    # Override the dispatch method to check permissions before processing the
request
    def dispatch(self, request, *args, **kwargs):
        # Get the instance of the listing being updated
        obj = self.get_object()
        # Check if the user making the request is the owner of the listing
        if obj.company != self.request.user:
            # If not, raise a PermissionDenied exception
            raise PermissionDenied
        # If the user is the owner, continue with the request processing
        return super().dispatch(request, *args, **kwargs)

# Define a class for deleting a listing, inheriting from LoginRequiredMixin and
DeleteView
class ListingDeleteView(LoginRequiredMixin, DeleteView):
    # Specify the model that this view is deleting
    model = Listing
    # Specify the template used for rendering the delete confirmation page
    template_name = 'Listing_delete.html'
    # Specify the URL to redirect to after successful deletion
    success_url = reverse_lazy('listing_list')
    # Specify the URL to redirect to for login if the user is not authenticated
    login_url = 'login'

    # Override the dispatch method to check permissions before processing the
request
```

```
    def dispatch(self, request, *args, **kwargs):
        # Get the instance of the listing being deleted
        obj = self.get_object()
        # Check if the user making the request is the owner of the listing
        if obj.company != self.request.user:
            # If not, raise a PermissionDenied exception
            raise PermissionDenied
        # If the user is the owner, continue with the request processing
        return super().dispatch(request, *args, **kwargs)



class ListingCreateView(LoginRequiredMixin,CreateView):
    model = Listing
    template_name = 'listing_new.html'
    fields = ('job_title', 'description')
    login_url = 'login'

    def form_valid(self, form):
        form.instance.company = self.request.user
        return super().form_valid(form)
```

Now, please attempt to revise the listing generated by other users.
http://127.0.0.1:8000/listings/2/edit/




You can’t do it.

A "403 Forbidden" status code is an HTTP response status code indicating that the server understood the request, but it refuses to authorize it.

## Conclusion

In conclusion, this chapter highlights the critical importance of Permissions and Authorization in ensuring robust data security. From defining permissions to establishing adaptive authorization strategies, we've explored the key elements of access control. Recognizing the dynamic nature of modern systems, readers gain insights into safeguarding sensitive information. This chapter serves as a foundation for implementing effective security measures in diverse contexts.

# Chapter 15: Conclusion

Congratulations on completing the “Projects for Beginners”! From starting at ground zero, we've successfully crafted five distinct web applications from the ground up, delving into all the crucial facets of Django, including templates, views, URLs, users, models, security, testing, and deployment. Armed with this knowledge, you're now well-equipped to embark on the journey of constructing your own contemporary websites using Django.

As with any newfound skill, diligent practice and application of your recently acquired expertise are essential. The CRUD functionality explored in our Blog and Newspaper projects is widely applicable across various web applications. Consider challenging yourself by creating a Todo List web application, utilizing the tools you've already mastered.

Web development is an expansive field, continuously evolving with new developments. As you embark on your journey, I recommend a methodical approach: initiate numerous small projects, progressively enhancing complexity, and exploring novel concepts. Attempting to construct a production-ready version of a complex platform like Twitter as your immediate next project may prove overwhelming and hinder the learning experience.

## Feedback

After completing the entire book, I'm eager to receive your feedback. What aspects did you enjoy or find challenging? Are there specific areas you found difficult? Additionally, I welcome suggestions for new content. Feel free to reach out to me at singhaditya1507@gmail.com. I look forward to hearing your thoughts!

## Also by Aditya Dhandi

- **Python Course - Learn Python From Scratch** :