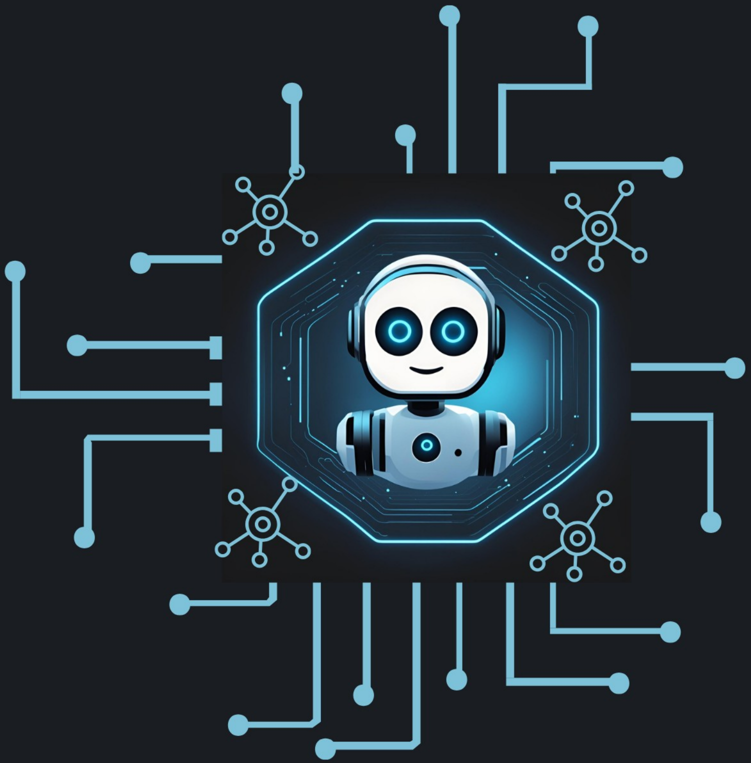
BUILDING A BOT PLATFORM WITH DJANGO



STEP-BY-STEP GUIDE TO BUILDING A BOT BUILDING PLATFORM FROM SCRATCH

Building a Bot Platform with Django

A comprehensive step-by-step guide to building a bot creation platform with Django,including integration with Bot Telegram and WhatsApp.

Zaenal Arifin

Building a Bot Platform with Django

Zaenal Arifin

Editor :
Zaenal Arifin

Cover Design :
Zaenal Arifin

Size :
***117963 Kata 752 Halaman.* Uk: 14x20 cm**

ISBN :
No ISBN

First Edition :
10/02/24

PUBLISHER AigoreTech

Website: www.aigoretech.my.id
E-mail: admin@aigoretech.my.id

Foreword

Welcome to this eBook!

We are delighted that you have chosen to read this work. This eBook is designed to provide a clear and practical understanding of the topic we discuss, without requiring complex technical details.

In this eBook, we have structured the material in a clear and organized manner to help you grasp the key concepts and their practical applications. Whether you are a beginner just starting your learning journey or someone looking to deepen your understanding in this field, we hope this eBook meets your needs.

We hope this material proves helpful and supports you on your learning journey. If you have any feedback or questions, please don’t hesitate to contact us.

Zaenal Arifin

Table of Contents

INTRODUCTION

Welcome to the World of Bot Development

Welcome to the eBook that will guide you through a deep and rewarding journey into the world of bot development. In recent years, bots have become increasingly important tools in various fields, from customer service to business automation. They not only simplify everyday tasks but also open new opportunities for individuals and companies to interact with their users in more efficient and personalized ways.

However, building a robust bot platform is no easy task. Many beginner developers often feel overwhelmed by the complexity of the technologies involved. This is where Django, a powerful and flexible web framework, comes in to provide an elegant and efficient solution. Django allows you to quickly build complex and scalable web applications without sacrificing quality or performance.

Why Django?

Django is not only one of the most popular web frameworks but also one of the most reliable. With a clear architecture and comprehensive documentation, Django is designed to assist developers of all skill levels—from beginners to experts—in building secure, scalable, and maintainable web applications. Django is used by some of the largest technology companies in the world, including Instagram and Pinterest, for very good reasons: it is a tool that enables big ideas to be realized quickly and effectively.

In this book, you will learn how to build a bot platform from start to finish using Django. This step-by-step guide will help you understand not only how Django works but also how you can leverage its power to create truly useful and valuable solutions.

What is a Bot?

A bot, or software robot, is a program designed to automate specific tasks within an application or on the internet. Bots are often used to perform routine tasks that can be pre-programmed, such as responding to inquiries, managing user interactions, or processing data automatically. These bots are incredibly beneficial in various scenarios, particularly for quick and efficient interactions with users without requiring direct human intervention.

In the context of web applications, bots often function as **chatbots** that automatically interact with users through chat interfaces, such as on Telegram or WhatsApp. Additionally, bots can operate behind the scenes to manage and process data automatically without direct user interaction.

Example Bots: Telegram and WhatsApp

Telegram Bot

Telegram bots are among the most popular types of bots today. Telegram provides a flexible platform for developers to create bots that can interact with users through chat conversations. Telegram bots can be used for various purposes, such as:

- ***Customer Service Chatbots***: Bots that automatically respond to customer inquiries, provide product information, or assist users in resolving issues.
- ***Reminders and Notifications***: Bots that send reminders to users about events, tasks, or important occurrences, as well as automated notifications.
- ***Task Automation***: Bots that execute specific tasks based on user instructions, such as monitoring product prices, providing weather reports, or executing scheduled processes.

A real-world example is a bot that can manage Telegram groups, moderate conversations, delete spam messages, or even conduct quizzes for users.

WhatsApp Bot

WhatsApp bots, on the other hand, are also very beneficial for businesses and organizations to interact with users through the WhatsApp platform. One popular service provider for building bots on WhatsApp is **Twilio**, which allows developers to create bots that connect directly with their WhatsApp numbers. The functionalities of WhatsApp bots include:

- ***Customer Support***: Bots that respond to common customer inquiries, provide order statuses, or guide users through various processes directly on WhatsApp.
- ***Notifications and Confirmations***: Bots that send notifications such as order confirmations, shipping statuses, or appointment reminders.
- ***Ordering and Payments***: Some businesses use WhatsApp bots to accept orders and process payments automatically, providing a quicker and easier shopping experience for users.

For instance, a bot on WhatsApp can be used by an online store to receive orders and automatically update customers on their shipping status.

Uses of Bots in Web Applications

1. ***User Interaction***: Bots can provide automatic responses to users, such as answering frequently asked questions (FAQs), providing guidance, or conveying important information quickly and efficiently.
2. ***Task Automation***: Bots can perform repetitive tasks, such as sending emails, providing reminders, or processing transactions, without requiring human intervention. This increases operational efficiency.
3. ***Data Processing***: Bots can gather, analyze, and present data from various sources, assisting in better decision-making. For example, a bot can monitor sales trends and provide regular reports to users.

Project Overview: Bot Platform

This project aims to build a platform that enables users to create and manage their bots independently through a simple yet powerful web interface. The platform will include several key features as follows:

1. ***Home (Landing Page)***: The main page welcoming new users and providing options for login or registration. This page will also provide a brief overview of the platform and its benefits.
2. ***Registration and Login Features***: An authentication system that allows users to create accounts, log in, and access their personal dashboards. This process is equipped with security validation to ensure that only legitimate users can access the platform.
3. ***Bot Management Dashboard***: After logging in, users will be directed to a dashboard that serves as a control center. Here, they can view general information about their bots, statistics, and recent activities.
4. ***Create Bot***: A feature that allows users to create new bots. Users can select a platform (e.g., Telegram or WhatsApp), then enter tokens or other specific platform information. The bot creation process is made easy with an interactive interface and clear instructions.
5. ***Manage Bot***: This feature allows users to view a list of all bots they have created. They can see bot details, bot status (active/inactive), and perform actions such as editing or deleting the bot.
6. ***Edit Bot***: Users can edit information for created bots, such as bot names, tokens, or other settings related to the

chosen platform. Each change will be saved and updated in real-time.

7. ***Add Command***: This feature allows users to add new commands to their bots. These commands can be automatic responses that will trigger when specific users send messages or instructions.
8. ***Edit Command***: In addition to adding commands, users can also edit existing commands. This allows them to update responses or the logic behind each command as needed.
9. ***Edit Profile***: Users also have the option to update their profile information, such as name, email address, and password. This feature ensures users have full control over their personal information.
10. ***Webhook and Bot Integration***: This platform also allows integration with external platforms through webhooks. This feature enables bots to receive and respond to data in real-time, ensuring seamless interaction between bots and external applications.

By leveraging Django as the core framework, this project is designed to provide a flexible, secure, and user-friendly solution. Django will handle data management and authentication systems, while the power of Python will be used to manage bot logic and webhook integration. This platform offers high scalability, allowing developers to add more features and bot platforms in the future.

About This Book

This book is specifically designed to help you build an advanced **bot creation platform** using the Django framework. In today's digital era, bots have become essential tools across various industries, ranging from customer service and marketing to business process automation. As the demand for more efficient and measurable solutions grows, building a flexible and scalable bot platform becomes increasingly relevant. This book will guide you through every step of the development process, from planning to final implementation.

Purpose of This Book

The main goal of this book is to **provide a practical** and **easy-to-understand guide**, even for those who are new to Django or web development in general. You will be guided step-by-step in building a Django-based web application that can be used to create, manage, and run bots using **webhooks**. Additionally, this book is designed to help you develop a deep understanding of various aspects of Django-based application development, from database management to interactive user interfaces.

Each chapter in this book focuses on critical parts of bot platform development, with instructions designed to take you from a basic level to a project that can be used in production. You will not only learn coding techniques but also the principles of good software development, such as version control, testing, and security implementation.

Who Should Read This Book?

This book is suitable for **beginners** who want to learn Django while building real applications. It not only covers the fundamentals of Django but also provides **practical use cases** that will help you understand how Django can be used to build complex and functional applications. While the primary focus of this book is on bot platform development, the concepts and skills learned can easily be applied to other Django projects.

This book is designed for beginner developers who want to deepen their knowledge of Django and how this framework can be used to build bot platforms. However, it is also suitable for experienced developers who want to learn how to integrate bots into their web applications.

This book targets:

- ***Django Beginners:*** Those who are just starting with Django and want to understand how this framework works through real projects.
- ***Developers Who Want to Learn Practically:*** Those who prefer to learn by diving directly into projects rather than just reading theory.
- ***Anyone Interested in Creating Bot-Based Applications:*** Whether for personal or professional needs.

You don't need to be an expert before starting this journey. All you need is a basic knowledge of Python and a passion for learning. Step by step, we will cover every aspect of bot platform development, from setting up the environment to deployment on a production server.

If you are a beginner web developer or someone looking to expand your skills in **bot development** and **webhook integration**, this book is the right choice. You do not need deep experience with Django or bot development beforehand, as this book will cover everything from the basics to advanced stages in a user-friendly manner.

Approach of This Book

This book employs a **straightforward** and **project-based** approach. Each chapter covers essential components of the development process, starting from setting up the working environment, Django project structure, to building key features such as the bot creation page, bot management, and webhook integration. You will learn how to use **Bootstrap** to enhance the application's appearance, as well as how to manage **database models** to store bot and user information.

Additionally, this book teaches good software development principles, including dependency management, testing, and **web application security**. You will be guided to understand how to secure bot tokens, how to use Django to manage user authentication, and how to avoid common pitfalls in bot development that interacts with external platforms like Telegram and WhatsApp.

Each feature built will be explained in detail, with code accompanied by explanations to help you understand every step taken. With this approach, you will not only learn how to create a functioning application but also grasp the **concepts behind it**, enabling you to apply this knowledge to other projects in the future.

Content Presented in This Book

This book consists of several chapters focusing on key topics, including:

- ***Introduction to Django and Project Setup:*** Discussing the fundamentals of Django, setting up the working environment, and project structure.
- ***Building the Bot Creation Page:*** Teaching how to create an interactive user interface for creating and managing bots.
- ***Bot Management and Commands:*** Introducing bot management features such as editing, deleting, and adding commands to the created bots.
- ***Webhook Integration:*** Discussing how to connect the bot platform with external platforms like Telegram and WhatsApp using webhooks.
- ***Security and Token Validation:*** Covering security aspects in bot development, including how to protect tokens and ensure that only valid tokens can be used.
- ***Testing and Deployment:*** Guidelines on how to effectively test the application and deploy it to a production server.

What You Will Gain After Reading This Book

After completing this book, you will have a **deep understanding** of how to build functional and complex Django applications. Not limited to bot creation, the skills you learn will encompass various aspects of web development, such as database management, interactive user interface design, API integration, and implementing security in web applications.

In addition, you will have a fully functional bot platform that you can further develop or use as a foundation for future projects. You will also become familiar with best practices in software development, enhancing your abilities as a more professional and experienced developer.

What We Will Build

In this final project, we will build a **complete and sophisticated bot creation and management platform**, allowing users to easily create, manage, and monitor their bots through a user-friendly interface. This platform not only provides tools for bot creation but also offers full control through a powerful **admin panel** and an interactive **user dashboard**. By leveraging Django as the backend framework, this project will integrate multiple bot services like **Telegram** and **WhatsApp**, while also enabling future platform development.

This platform will focus on security, scalability, and ease of use, allowing users to interact with and manage bots without requiring in-depth technical knowledge. Below are the main features of this project:

1.Registration and Login Page

Users need to register and log in to the platform using a secure and intuitive registration and login page. These features will include additional security layers to ensure that only legitimate users can access the platform. The features included are:

- **Registration form** for new users, with strict data validation to ensure the information entered is accurate and secure.
- **Login form** for registered users, with secure authentication mechanisms to prevent unauthorized access.
- **Password recovery feature**, allowing users who forget their passwords to reset them through email verification.
- **Email verification** to ensure that created accounts are legitimate and to provide additional security when users log in.

2.User Dashboard Page

After successfully logging in, users will be directed to their personal dashboard page. This dashboard serves as a control center where users can easily manage all aspects of their bots. The main features on the dashboard page include:

- ***Bot Management:*** Users can view a list of bots they have created, add new bots, or edit and delete existing ones. Each bot will be displayed with basic information such as name, platform, and status (active or inactive).
- ***Bot Template Options:*** The platform provides various ready-to-use bot templates, enabling users to get started quickly without needing to write code from scratch. Users can choose and customize bot templates according to their needs.

- ***Bot Statistics:*** The dashboard will display important statistics related to bot performance, including the number of users interacting, messages sent, and error reports. This information will be presented in graphs and tables for easier analysis.
- ***Additional Features:*** Users can also manage bot commands, such as adding new commands, editing existing ones, or deleting them. All of this can be done through an intuitive interface.

3.Admin Panel

This platform will be equipped with a powerful admin panel to assist administrators in managing users, bots, and other features. The admin panel functions as the overall management center for the platform, with exclusive access for administrators to configure various important elements. Features provided in the admin panel include:

- ***User Management:*** Administrators can add, edit, or delete registered users on the platform. Additionally, admins can view user activity, such as when they last logged in or what activities they have performed.
- ***Access Management:*** Admins have full control over user access rights. They can grant privileges to specific users, such as admin or moderator access.
- ***Bot Management:*** Admins can monitor all bots created by users, as well as add additional features to specific bots. Admins also have the ability to delete bots that violate platform policies.
- ***Bot Template Management:*** Admins can add, edit, or delete bot templates available on the platform. This

allows admins to add new options that users can choose from when creating bots.

4.User Panel

The user panel is designed to provide an optimal and intuitive user experience, allowing users to interact with the platform easily. This panel contains several key features that support users' daily activities on the platform:

- ***Bot Creation and Management:*** Users can quickly create new bots through an easy-to-understand wizard. Once a bot is created, users can manage every aspect of that bot, such as changing settings, adding new commands, or even deleting bots that are no longer needed.
- ***Access to Bot Templates:*** Users can choose from various bot templates provided by the platform. Each template can be customized based on user preferences, both in terms of functionality and appearance.
- ***Personal Dashboard:*** Each user will have a personal dashboard displaying all information related to the bots they manage, such as the number of user interactions, the number of messages sent by the bot, and recent activities. This gives users clear insights into their bots' performance and steps they can take to enhance it.

By following the guidelines in this book, you will learn how to build a functional, secure, and scalable bot platform using Django. In addition to building a user-friendly interface, you will also be taught to manage the technical aspects behind the scenes, such as authentication, token validation, bot management, and implementing webhooks for seamless bot integration with external platforms. After completing this project, you will have the skills and understanding necessary to develop similar or even more complex platforms in the future.

Let's Get Started!

The world of bots awaits you. With Django as your main tool, let’s embark on this adventure and see how far you can go. There will not only be challenges along the way, but also many opportunities to learn and grow. I am confident that by the end of this book, you will see Django not just as a framework, but as a powerful ally in your development journey.

Chapter 1 - Introduction to Django

In this chapter, we will explore the fundamentals of Django, a powerful web framework that is highly popular among developers. We will start by understanding what Django is, including a brief history of its development and why it has become the framework of choice for many developers worldwide. This chapter will introduce the various key features of Django that set it apart from other frameworks, such as Object-Relational Mapping (ORM), an automatic Admin Panel, a flexible templating system, and its support for high security and performance.

Additionally, we will delve into the advantages of Django, such as its ability to speed up the development process, ease of integration with various databases, and its structured architecture. To facilitate understanding, we will also discuss the Model-View-Template (MVT) architectural concept that underpins Django in building web applications.

After grasping the foundations of Django, we will explore the main components that make up a Django project, such as apps, models, views, templates, and URL routing. Ultimately, this chapter will provide a clear overview of why Django is the right choice for web application development, from small to large scale.

Thus, this chapter will serve as an important foundation for all subsequent technical discussions, and we will be ready to start building efficient and scalable Django-based applications.

1.1 What Is Django?

Django is a web framework based on Python, designed to facilitate the development of complex and dynamic web applications. Developed by the Django Software Foundation, this framework offers a variety of advanced and practical features to speed up the development process.

Generally, Django uses the Model-View-Template (MVT) architectural concept, which allows for the separation of business logic, data, and presentation. This separation makes it easier to develop larger projects, as each part can be broken down and handled independently.

The strength of Django lies in its "batteries included" approach, where it provides many built-in features, such as an authentication system, URL management, and Object-Relational Mapping (ORM) for managing databases. With Django, we can easily build complex web applications without having to worry about many technical details, such as security, input validation, and session management, as all of these are taken care of by the framework.

Here are some key points about Django:

Full-Stack Web Framework: Django is a full-stack framework, meaning it encompasses various components necessary for building web applications, such as database management systems, URL routing, form processing, and templating systems for creating user interfaces.

Python-Based: Django is built using the Python programming language, known for its clean syntax and ease of use. Python allows developers to write efficient and readable code, making it a popular choice in web development.

Focus on Security: Django includes built-in security features that help protect applications from various common threats, such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). This provides extra protection and reduces security risks.

DRY Principle (Don’t Repeat Yourself): Django follows the DRY principle, which means that the same code does not need to be written multiple times. This helps reduce code duplication and improves development efficiency.

Automatic Admin System: One of Django's standout features is its automatic admin system. After defining data models, Django can automatically generate an admin interface that allows you to easily manage application data.

MVC Architecture: Django uses the Model-View-Controller (MVC) architecture, although the terminology used in Django is Model-View-Template (MVT). In this architecture:

- ***Model***: Defines the application's data structure and interacts with the database.
- ***View***: Handles application logic and processes requests from users.
- ***Template***: Provides a way to display data in HTML format that can be viewed by users.

Community and Documentation: Django has an active developer community and excellent documentation. This makes it easy to seek support, find solutions to problems, and learn more about the framework.

With these features, Django is an excellent choice for building various types of web applications, from simple websites to complex and fully-featured applications. In this book, you will learn how to leverage Django to create a bot creation platform from scratch, with a practical approach that is easy to follow.

1.2 Brief History Of Django

Django did not just emerge as the web framework we know today. The history of Django began in 2003, when a team of developers at the Lawrence Journal-World media company, based in Kansas, USA, faced the challenge of managing multiple news websites. These sites required quick and frequent updates, and the development team needed a way to manage content efficiently. The existing frameworks at that time did not meet their needs, so they started building their own framework to speed up the development process.

Initially, this framework did not have an official name and was only used internally. However, due to the efficiency it offered—particularly in terms of content management and dynamic website creation—this framework quickly became more important to the team. In 2005, it was officially released as open-source and named **Django**, after the legendary jazz guitarist Django Reinhardt. The name reflects the framework's fast, agile, and flexible nature, reminiscent of Django Reinhardt's famous guitar playing style.

Since its release, Django has continued to evolve and attract the attention of many developers worldwide. With a growing community, Django has seen continuous improvements in features and security. One of the biggest changes in Django's evolution is its ability to handle increasingly complex projects, both in terms of scalability and flexibility. Each major release of Django not only brings performance enhancements but also new features that address the needs of modern web development.

A significant milestone in Django's history was the release of Django 1.0 in September 2008. This was a major historical landmark as it signified the stability of the framework with a mature API, comprehensive documentation, and a long-term commitment to backward compatibility support. Over time, Django also began supporting the latest technologies, such as Python 3, after Python 2 was phased out by the community.

Subsequent versions of Django have increasingly emphasized ease of use, security, and scalability. One of Django's main strengths is its focus on security. Django consistently protects developers from common mistakes that often occur in web development, such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). These features make Django highly reliable for building secure web applications from the start.

Additionally, Django has been adopted by various large technology companies and is used in production environments. Notable companies using Django for their application development include Instagram, Pinterest, and Mozilla. The usage of Django by these companies demonstrates that this framework can handle large scales and high traffic volumes.

One of the primary reasons for Django's success is its excellent documentation. From the beginning, Django developers recognized that documentation is key to the framework's adoption by the community. The available documentation covers not only basic tutorials but also advanced guides for more complex features. This enables both beginners and experienced developers to use Django effectively.

Today, Django remains one of the most popular web frameworks in the world. With its latest releases continually introducing modern features, Django is committed to staying relevant in the ever-evolving world of web development. For instance, integration with asynchrony in Django 3.x allows developers to build more efficient applications for handling concurrent requests.

With a strong foundation, a large community, and support from major companies, Django continues to be a top choice for web application development across various industries, from media and e-commerce to financial technology (fintech).

If we work on a project like **`platform_bot`**, we can leverage all the features that have been developed over the years, such as flexible URL management, the use of ORM to interact with databases, and a powerful templating system.

Django is designed with the core philosophies of "Don't Repeat Yourself" (DRY) and "Explicit is Better Than Implicit." These philosophies enable developers to write clean, efficient, and maintainable code.

1.3 Key Features Of Django

One of the main reasons Django is a choice for many developers is its various built-in features that facilitate the web application development process. These features are designed to speed up development without compromising quality and security. In this section, we will discuss some core features that make Django stand out, such as ORM (Object-Relational Mapping), an automatic admin panel, and a routing system.

1.3.1 ORM (Object-Relational Mapping)

Django comes with a powerful and efficient built-in ORM that allows us to interact with databases without manually writing SQL queries. This ORM connects Python models with database tables, enabling us to manage data in our applications using Python objects. The ORM also allows us to perform complex database operations with simple and intuitive syntax.

For example, we can define a model to manage bot data in the `platform_bot` project. This model will be translated into the corresponding database table.

```
# platform_bot/models.py
from django.db import models

class Bot(models.Model):
    # Defining model fields
    name = models.CharField(max_length=100)  # Bot
name
    platform = models.CharField(max_length=50)  # Bot
platform (e.g., Telegram, WhatsApp)
    created_at =
models.DateTimeField(auto_now_add=True)  # Bot
creation date

    def __str__(self):
        return self.name  # Return the bot's name
when called
```

After defining the model, we need to create and apply migrations to create the corresponding tables in the database:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

With Django ORM, we can perform database operations in a straightforward manner. For example, to create a new entry in the `Bot` table, we simply write:

```
# Create a new entry for Bot
new_bot = Bot(name="ChatBot", platform="Telegram")
new_bot.save()  # Save the new entry to the database
```

We can also easily perform other operations like retrieving data, updating, or deleting entries without needing to write SQL queries.

1.3.2 Automatic Admin Panel

One of the unique features that makes Django very appealing is the automatic admin panel generated from the models we create. With just a few simple configurations, Django can produce an administrative interface to manage data in our applications, which is incredibly helpful in the development and management processes.

To enable the admin panel for the `Bot` model we created earlier, we just need to add it to the `admin.py` file:

```
# platform_bot/admin.py
from django.contrib import admin
from .models import Bot

# Registering the Bot model with the admin panel
admin.site.register(Bot)
```

After that, we can access the admin panel by opening `http://127.0.0.1:8000/admin/` in a browser and logging in using the superuser account we created earlier. There, we will see an interface that allows us to create, edit, or delete entries in the `Bot` table without writing additional code for CRUD operations.

This admin panel is very flexible and can be further customized if needed. We can easily add search features, filters, or modify how data is displayed in the admin panel.

1.3.3 Security

Security is one of the most important aspects of web application development, and Django consistently prioritizes security as a key focus. Django comes equipped with various built-in security features that protect applications from common threats on the web. One of the main features is protection against SQL Injection, where Django's ORM ensures that all queries to the database are safe and protected from external manipulation.

In addition, Django also protects against Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). Every form we create in Django automatically includes a CSRF token, which prevents attacks attempting to exploit user sessions to perform actions without their knowledge. To protect user data, Django also supports password hashing using secure algorithms.

For instance, to enable CSRF protection on the forms we create, Django automatically adds this token in the template file:

```
<form method="post">
    {% csrf_token %}  <!-- This token prevents CSRF
attacks -->
    <!-- Form contents -->
</form>
```

Django also ensures that developers do not have to worry about issues such as Clickjacking and Session Hijacking, as these features are managed automatically.

1.3.4 Templating System

Django has a very powerful templating system that allows us to build web page views dynamically. This system supports the use of templates to separate programming logic from the presentation. Thus, frontend and backend development can be done separately, and templates can be reused in multiple places.

Django's templating system supports template inheritance, allowing us to create a base template that can be extended by other templates. For example, we can create a `base.html` file as the main framework, and other templates like the bot page or login page can extend it. Here is a simple example of the base template:

```
<!-- templates/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Platform Bot</title>
</head>
```

```
<body>
    <header>
        <!-- Header content -->
    </header>
    <main>
        {% block content %}  <!-- Block that will be
extended by other templates -->
        {% endblock %}
    </main>
    <footer>
        <!-- Footer content -->
    </footer>
</body>
</html>
```

Then, for another page, we can use template inheritance:

```
<!-- templates/bot_list.html -->
{% extends 'base.html' %}

{% block content %}
<h1>List of Bots</h1>
<ul>
    {% for bot in bots %}
        <li>{{ bot.name }} - Platform:
{{ bot.platform }}</li>
    {% endfor %}
</ul>
{% endblock %}
```

With this system, we can maintain consistency across pages and save time in writing template code.

1.3.5 Flexible URL Routing

Another important feature of Django is its very flexible routing system. Django uses URL patterns that allow us to define URLs for each view or page in our application in a highly structured way. Each URL entered by the user will be mapped to the corresponding view.

To define a URL pattern, we need to create an entry in the `urls.py` file. For example, if we want to display a list of all existing bots, we can add a view and URL for that purpose.

```
# platform_bot/views.py
from django.shortcuts import render
from .models import Bot

# Display the list of bots
def bot_list(request):
    bots = Bot.objects.all()  # Retrieve all bots
from the database
    return render(request, 'bot_list.html', {'bots':
bots})  # Render the page with bot data
```

Next, we add a URL pattern that links the view to the corresponding URL:

```
# platform_bot/urls.py
from django.urls import path
from .views import bot_list

urlpatterns = [
    path('bots/', bot_list, name='bot_list'),  # URL
to display the list of bots
]
```

With this routing system, we can easily create clean and understandable URLs for each page in our application. Django also supports more complex URL patterns, such as dynamic URLs that accept parameters, e.g., `bots/<int:id>/` to display details of a specific bot.

This routing system works well with Django's templating system, allowing us to build dynamic user interfaces based on data sent from views.

1.3.6 Strong Community and Documentation

One of the main factors that contribute to Django's continuous growth is its very active global community. Django has a solid community where developers from around the world contribute to the development of new features, fix bugs, and provide support for other users. Django is also known for its comprehensive and clear documentation. This documentation covers everything from basic introductions to more complex concepts.

Django holds annual conferences like **DjangoCon**, which serve as a gathering place for Django developers to share knowledge, experiences, and best practices. This community is also very responsive in addressing security issues, so if vulnerabilities are found, security patches are usually released quickly.

For beginner developers, the official Django documentation is one of the most user-friendly resources available. The built-in Django tutorials allow us to build applications from scratch, covering everything from installation to deployment to production servers.

1.3.7 Scalability and Performance

Django is designed to support large-scale web applications while remaining optimized for high performance. With an easily implementable caching system and support for the use of large-scale databases, Django enables us to efficiently handle millions of users and requests. Django supports various caching mechanisms, such as memory-based caching or file-based caching, to speed up application response times.

For example, we can enable caching at the view level using a decorator:

```
# platform_bot/views.py
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)  # Cache for 15 minutes
def bot_list(request):
    bots = Bot.objects.all()
    return render(request, 'bot_list.html', {'bots':
bots})
```

With this caching feature, Django will store the results of specific requests to reduce server load and speed up response times for users.

Django's scalability is also evidenced by its use by large companies like Instagram and Pinterest, which successfully run large-scale applications based on Django.

Django not only provides the tools you need to build web applications but also offers guidelines and best practices to ensure your application is secure, scalable, and easy to maintain. As technology advances, Django continues to evolve and offers new features to support modern developer needs.

Django's key features, such as ORM, automatic admin panel, and flexible routing system, make it one of the most powerful and efficient web frameworks for web application development. Django not only helps us write clean and structured code but also enables rapid and secure development.

1.4 Django Advantages

Django is a highly popular framework among developers due to the many advantages it offers. These advantages make Django the top choice for various types of projects, ranging from small-scale applications to large applications used by millions of users. In this section, we will discuss several key reasons why Django is so sought after, including its speed, security, and scalability.

1.4.1 Rapid Development

One of the main reasons many developers choose Django is its speed in helping to build applications. Django is designed to accelerate the development process by providing many built-in features that help cut down on coding time. For example, the powerful Django ORM allows us to work with databases without having to write complex SQL queries, while the automatic admin panel simplifies data management without needing to build an admin panel from scratch.

Additionally, Django provides **reusable components** that can be used repeatedly across different parts of the application, such as templates, forms, and views. This makes it easier to reduce code redundancy and speed up the development of new features.

Django follows the **DRY (Don't Repeat Yourself)** principle, which means we can write more efficient and maintainable code.

For example, when developing a bot application in the **platform_bot** project, we can create a view to display a list of bots and use it in many places with minimal modifications:

```
# platform_bot/views.py
from django.shortcuts import render
from .models import Bot

# View to display the list of bots
def bot_list(request):
    bots = Bot.objects.all()  # Retrieve all bots
from the database
    return render(request, 'bot_list.html', {'bots':
bots})  # Render the page with bot data
```

With Django, we can build complex applications in a shorter time without sacrificing functionality.

1.4.2 Integrated Security

Security is a top priority for Django, and this is one of the main reasons why this framework is often chosen for large-scale web applications. Django automatically handles many security aspects, such as **CSRF protection**, **XSS protection**, **SQL Injection protection**, and secure password hashing. All of these features are enabled by default, so developers don't have to worry about the basic security of their applications.

For instance, when we create a form in Django, a **CSRF** token will automatically be included to prevent attacks that exploit user sessions:

```
<form method="post">
    {% csrf_token %}  <!-- CSRF token is
automatically generated by Django -->
    <!-- Input form -->
</form>
```

With Django, developers can focus on business logic without having to worry too much about common security threats that often target web applications.

1.4.3 Scalability

Django has proven to be highly scalable, meaning that applications built using Django can easily grow to handle hundreds to millions of users. This is one of the reasons why Django is chosen by large companies like Instagram and Pinterest. With its modular architecture and support for various caching and database solutions, Django can be optimized for higher performance as the number of users increases.

Django supports integration with various types of databases, from lightweight SQLite to more robust options like PostgreSQL and MySQL for large-scale applications. Django also supports **horizontal scaling** by allowing the application to run on multiple servers to handle increased traffic.

We can also use Django's built-in **caching** feature to improve application performance by reducing server load. Caching is especially useful for pages or operations that frequently generate the same data, so they don't need to be processed repeatedly.

For example, to enable view-level caching, we can use Django's `cache_page` decorator:

```
# platform_bot/views.py
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)  # Cache the view for 15 minutes
def bot_list(request):
    bots = Bot.objects.all()  # Retrieve bot data
from the database
    return render(request, 'bot_list.html', {'bots':
bots})  # Render the page with bot data
```

In this way, applications built with Django can handle more requests without sacrificing performance.

1.4.4 Structured Architecture

Django is designed with a highly structured architecture that separates **models (M)**, **views (V)**, and **templates (T)**. This structure is known as the **Model-View-Template (MVT)** pattern. By using this pattern, we can maintain order in application development, making the code easier to maintain and scale as the project grows.

In MVT architecture, the Model is responsible for managing data and interacting with the database, the View handles the application logic and request processing, while the Template controls how data is presented to the user. This separation allows development teams to work in parallel on different parts of the application without interference.

For example, in the **`platform_bot`** application, we could have the following MVT structure:

- ***Model*** in `platform_bot/models.py`, which defines bot data.
- ***View*** in `platform_bot/views.py`, which processes requests to display a list of bots.
- ***Template*** in `platform_bot/templates/bot_list.html`, which displays bot data on the webpage.

With this structure, we can separate the application logic from the presentation layer, making it easier to manage the code and ensuring that the application remains modular and scalable as the project grows.

1.4.5 Automatic Admin Interface

One of the most interesting features of Django is its automatically generated **admin interface**. When we create a model in Django, the framework automatically generates an admin panel that allows us to easily manage data without writing additional code for the admin interface. This feature is highly beneficial, especially for rapid development and managing internal applications.

Django's admin panel allows us to **add**, **edit**, and **delete data** directly from the provided interface. We can also customize the admin panel's appearance as needed, by adding specific models or configuring how the data is displayed.

For example, in the **`platform_bot`** project, we can register the bot model in the admin panel with just a few lines of code in the `admin.py` file:

```
# platform_bot/admin.py
from django.contrib import admin
from .models import Bot

# Register the Bot model in the admin panel
admin.site.register(Bot)
```

After this step, we can access the admin panel through the `/admin` URL and directly manage bot data without having to build a custom interface.

1.4.6 Support for Various Databases

Django supports a variety of **databases**, including **SQLite**, **PostgreSQL**, **MySQL**, and **Oracle**. This capability to work with multiple database types makes Django highly flexible and allows developers to choose the database that best suits their project needs. During development, we might use **SQLite**, which is lightweight and easy to set up, but as the application grows, we can easily switch to a more robust database like **PostgreSQL** or **MySQL**.

To change the database, simply configure the settings in the `settings.py` file:

```
# platform_bot/settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',  #
Use PostgreSQL
```

```
        'NAME': 'platform_bot_db',
        'USER': 'postgres_user',
        'PASSWORD': 'your_password',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

With a few changes to this configuration, Django will automatically adjust itself to work with the new database.

1.4.7 Built-in Features for Web Development

Django offers a wealth of **built-in** features that significantly simplify web development. From **routing** systems to **session** management, Django provides all the necessary tools for building complete web applications. For example, Django includes a built-in **form** system that makes it easy to create, validate, and process data from HTML forms.

Additionally, Django includes various **middleware** features, such as **authentication** and **authorization** management, which simplify user session management. This allows developers to focus on the application’s logic while Django handles the more complex technical details.

Moreover, Django is equipped with a **caching** system that can enhance application performance by temporarily storing data. All of these features are provided without excessive configuration or the need for additional external libraries.

1.4.8 Ease of Testing

Django offers **built-in** support for testing, allowing developers to easily write and run **unit tests** to ensure that application features function correctly. Testing is a crucial aspect of software development, and Django simplifies it by providing an integrated testing framework.

You can write test cases to ensure that views, models, and other functions work as expected. Django also provides tools for database testing, enabling **developers to test data** integrity during development.

For example, to test if the view displaying the list of bots works correctly, you can write a test case like this:

```python
# platform_bot/tests.py
from django.test import TestCase
from .models import Bot

class BotViewTests(TestCase):
    def test_bot_list_view(self):
        response = self.client.get('/bots/')  # Send
a request to the bot list view
        self.assertEqual(response.status_code, 200)
# Ensure the response status is 200 OK
        self.assertContains(response, 'Daftar Bot')
# Check if the text "Daftar Bot" is present on the
page
```

This test can be run using the following terminal command:

```
$ python manage.py test
```

With Django, you can automate testing and maintain the quality of the application throughout the development cycle.

1.4.9 Support for Large-Scale Application Development

Django is an excellent choice for large-scale application development due to its well-organized and clear structure. Django enforces separation between components such as models, views, and templates, which aids in code management and facilitates collaboration among team members.

Furthermore, Django supports **modular** application development, where each part of the application can be packaged as an **app**. This makes it easier to break down large application features into smaller, more manageable modules. For example, in the **platform_bot** project, you could create separate apps to manage bots, platform components, and so on, without mixing everything in one place.

By using Django, development teams can work on different parts of the application without conflicts, as Django provides clear structure and guidelines.

Django’s many advantages, from its incredible development speed to its robust security and proven scalability, make it a highly popular choice among developers. Django not only helps developers build applications quickly but also ensures that those applications are secure and capable of handling large numbers of users.

With these advantages, Django is a solid choice for web development, whether for small or large projects. In this book, you will leverage Django’s strengths to build an effective and efficient bot-building platform.

1.5 MVC Architecture In Django

One of the architectural concepts that serves as the foundation for many web frameworks is **Model-View-Controller (MVC)**. Django also adopts this approach, albeit with slight modifications. In Django, the MVC approach transforms into **Model-View-Template (MVT)**, where the main components of this architecture maintain a clear separation between business logic, data, and presentation.

At its core, both MVC and MVT aim to separate responsibilities within an application, keeping the code clean, modular, and easy to maintain. The difference is that in Django, the **Template** takes on the role typically held by the Controller in MVC. Django automatically handles many tasks that would normally be the **controller's** responsibility in other frameworks, such as mapping URLs to views and processing HTTP requests.

1.5.1 Model

The **Model** in Django represents the data and business logic of the application. It interacts directly with the database and is responsible for managing the data, including defining columns and relationships between entities in the database. The model determines how data is stored, modified, and retrieved from the database. Django provides an **ORM (Object-Relational Mapping)** that allows us to work with the database using Python objects instead of SQL.

Function: In Django, models are defined as classes in the `models.py` file. Each model class represents a table in the database, and each class attribute represents a column in that table. Django provides various tools for migrating the database based on these models.

Example: For instance, if you have a `User` model, this model will define attributes like name, email, and password, as well as methods for interacting with user data.

1.5.2 View

The **View** is the part of the application that handles application logic and determines how data is processed and sent to the user. Views are responsible for receiving HTTP requests, retrieving the necessary data from the model, and returning a response in the form of a web page (or another type of response, such as JSON). In Django, the view also serves as a bridge between the **model** and the **template**.

Function: In Django, views are defined as functions or classes in the `views.py` file. A view retrieves data from the model, processes it if necessary, and sends that data to the template for display to the user.

Example: For example, a `profile_view` might retrieve user data from the `User` model and then send that data to the `profile.html` template for display.

1.5.3 Template

Templates are the part of the application that handles presentation or display. Django uses templates to render data from views into HTML format, which will be shown to the user. Templates in Django utilize a simple template language, allowing us to write HTML with dynamic elements pulled from views.

Function: In Django, templates are defined in HTML files located in the `templates` directory. Templates use Django Template Language (DTL) syntax to insert data from views into the HTML.

Example: For instance, the `profile.html` template will specify how user data (such as name and email) is displayed in an easily readable HTML format.

1.5.4 How MVT Works Together

The **MVT** architecture works in a highly structured and clear manner within Django applications. The flow of the process can be described as follows:

1. ***Request:*** When a user visits a specific page in a Django application, an HTTP request is sent to the server. Django uses a **URL routing** system to map the requested URL to the appropriate view.

2. ***View:*** Once the request is received, Django calls the view responsible for handling that request. The view may interact with the model to retrieve data from the database.
3. ***Model:*** If the view needs data from the database, it requests that data through the **model**. The model manages business logic and ensures that the requested data complies with application rules.
4. ***Template:*** After the view obtains the required data, it will use a template to render the data in HTML format. The **template** takes the data processed by the view and displays it to the user in the designed interface.
5. ***Response:*** Django sends the **response** back to the user in the form of HTML or another format (such as JSON) through their browser.

For example, when a user visits the URL `/bots/` to see the list of bots:

- **URL routing** will map this request to the `bot_list` view.
- The **`bot_list` view** will retrieve bot data from the **`Bot` model**.
- That data will be sent to the **`bot_list.html` template**.
- The template will render the data into HTML, which will then be sent back as a response to the user's browser.

In this way, Django maintains a clear separation between data (model), application logic (view), and presentation (template), ensuring that the application is easy to maintain and manage.

By separating the application into Model, View, and Template components, Django allows developers to better manage and organize their code, as well as facilitate the maintenance and development of web applications.

1.6 Key Components Of Django

Django consists of various core components that work together to create functional and structured web applications. Each component has a specific role and supports modular and manageable application development. In this subsection, we will discuss the important components that form the foundation of the Django framework.

1.6.1 App

In Django, the term **app** refers to small parts of a project that have a specific function and are separate. Django is designed to support modular application development, meaning a Django project can consist of many separate but interrelated apps. Each app can manage one part of the overall project's functionality, such as an app for user management, an app for a blog, or an app for a booking system.

With this approach, Django apps are **very portable**, allowing us to move existing apps to other projects with minimal configuration changes. This supports high **reusability**, so the same code can be reused in different projects.

Creating a New App

Each Django project usually consists of several apps. To create a new app, we can use the following command in the terminal:

```
$ python manage.py startapp myapp
```

This command will generate the necessary directory structure and files for the app:

```
myapp/
    migrations/
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
```

Each file in this directory has a specific role:

- ***admin.py***: Used to configure the Django admin panel.
- ***apps.py***: Contains the configuration for the app.
- ***models.py***: Where we define models, which are representations of data in the database.
- ***views.py***: Contains views that connect application logic with presentation.
- ***tests.py***: Used to write unit tests for this app.

After creating the app, we need to add it to the Django project by including it in the `INSTALLED_APPS` list in the `settings.py` file:

```
# platform_bot/settings.py
INSTALLED_APPS = [
    # Built-in Django apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Our created app
    'myapp',  # The app we just created
]
```

By adding it to `INSTALLED_APPS`, Django will recognize the app and include it in the development cycle.

Modular Structure for Large Projects

In large projects, this modular application architecture is very beneficial. We can divide each part of the application into small modules, where each module handles a specific aspect of the project. For example, in the **`platform_bot`** project, we might have several apps like:

- ***auth_app***: To handle user authentication and authorization.
- ***bot_app***: To manage bots and related platforms.
- ***command_app***: To manage customizable bot commands by users.

By separating apps into distinct modules, each development team can work on one part without interfering with others. Django also allows developers to use third-party apps, which can be directly installed into the project without having to rewrite existing functionality.

Managing Django Apps

Managing apps in Django is done with the **`manage.py`** command. The `manage.py` file is a tool used to interact with the Django project, such as creating apps, running migrations, and managing the database.

Some common commands for managing apps include:

- Creating a new app:

```
$ python manage.py startapp app_name
```

- Performing database migrations for newly created models:

```
$ python manage.py makemigrations
```

- Applying migration changes to the database:

```
$ python manage.py migrate
```

For example, if we add a new model to the **`myapp`** application, we need to create a migration and apply it for the changes to reflect in the database.

Example of App Usage in a Project

For example, let's create a new app called **`bot_app`** in the **`platform_bot`** project. First, we will create the app using the terminal command:

```
$ python manage.py startapp bot_app
```

After creating the app, we will add it to the `settings.py` file:

```
# platform_bot/settings.py
INSTALLED_APPS = [
    # Built-in Django apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Our created app
    'bot_app',  # New app that manages bots
]
```

In `bot_app/models.py`, we can define a model that represents the bot data:

```
# bot_app/models.py
from django.db import models

class Bot(models.Model):
    name = models.CharField(max_length=100)
    platform = models.CharField(max_length=50)
    created_at =
models.DateTimeField(auto_now_add=True)
```

This model organizes the bot data that will be managed in the application. After creating the model, we can run migration commands to apply it to the database:

```
$ python manage.py makemigrations bot_app
$ python manage.py migrate
```

This app is now part of the project, ready for further development with additional views, forms, templates, and other features.

With this modular app approach, Django supports the development of large projects that are easy to manage. Each app can be developed and tested separately while still working together in a complete project.

1.6.2 Template

In Django, a **Template** is a component that handles the view or presentation aspects of a web application. Templates are responsible for converting data from views into a format that can be displayed in a browser, such as HTML. By using Django's template system, we can build dynamic user interfaces that allow the displayed data to change based on input and the status of the application.

Django Template System

Django uses a template system based on a simple yet powerful templating language. This system allows us to combine static HTML with dynamic elements pulled from views, resulting in interactive and responsive web pages. The template system enables us to separate presentation logic from business logic and data, thus keeping the code clean and organized.

Each template in Django contains a combination of HTML and template tags used to insert data from views. Template tags and filters allow us to perform operations such as loops, conditions, and data formatting directly within the template.

Template Structure

Templates are placed in a `templates` directory within the Django app or the project folder. The template directory structure can be customized, but generally, templates are grouped in subdirectories based on the app or type of view.

For example, if we have an app named **`bot_app`**, the template structure might look like this:

```
bot_app/
    templates/
        bot_app/
            bot_list.html
            bot_detail.html
```

In the `templates/bot_app/` directory, we can store templates used to display bot-related data, such as the bot list and bot details.

Creating and Using Templates

To create a template, we can make an HTML file in the templates directory and write regular HTML while adding the necessary template tags. For instance, we can create the `bot_list.html` template to display a list of bots as follows:

```
<!-- bot_app/templates/bot_app/bot_list.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Bot List</title>
</head>
<body>
    <h1>Bot List</h1>
    <ul>
        {% for bot in bots %}
        <li>{{ bot.name }} - {{ bot.platform }}</li>
        {% empty %}
        <li>No bots found.</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Here, `{% for bot in bots %}` is a template tag used to iterate through the list of bots sent from the view. The `{% empty %}` template tag provides an alternative if the list is empty, while `{{ bot.name }}` and `{{ bot.platform }}` are used to insert dynamic data into the template.

Using Templates in Views

Once a template is created, we need to connect it with a view that processes data and passes it to the template. For example, in the **`bot_app`** application, we might have a `bot_list` view used to render the `bot_list.html` template with bot data:

```
# bot_app/views.py
from django.shortcuts import render
from .models import Bot
```

```
def bot_list(request):
    bots = Bot.objects.all()  # Fetch all bot data
from the database
    return render(request, 'bot_app/bot_list.html',
{'bots': bots})
```

The `render` function in this view takes three arguments:

- `request`: The HTTP request object.
- `'bot_app/bot_list.html'`: The path to the template to be rendered.
- `{'bots': bots}`: The context containing data to be used in the template.

Django will fetch the `bot_list.html` template, replace the placeholders with the data sent from the view, and generate an HTML page that is sent as a response to the user.

Template Inheritance

Django also supports **template inheritance**, allowing us to create a base template and inherit or override specific parts of that template in child templates. This is very useful for consistency and maintaining the user interface across the application.

For example, we can create a base template `base.html` that contains the general HTML structure and then create specific templates that inherit from `base.html`:

```
<!-- bot_app/templates/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <title>{% block title %}My Site{% endblock
%}</title>
</head>
<body>
    <header>
        <h1>Website Header</h1>
    </header>
    <main>
        {% block content %}{% endblock %}
    </main>
    <footer>
        <p>Website Footer</p>
    </footer>
</body>
</html>
```

Then, we can create the `bot_list.html` template that inherits from `base.html` and fills in the specific content block:

```
<!-- bot_app/templates/bot_app/bot_list.html -->
{% extends 'base.html' %}

{% block title %}Bot List{% endblock %}

{% block content %}
<h2>Bot List</h2>
<ul>
    {% for bot in bots %}
    <li>{{ bot.name }} - {{ bot.platform }}</li>
    {% empty %}
    <li>No bots found.</li>
    {% endfor %}
</ul>
{% endblock %}
```

By using inheritance, we can maintain design consistency and make HTML code maintenance easier.

The Django template system enables developers to efficiently separate presentation logic from application logic. With the ability to use template tags and filters, as well as support for template inheritance, Django provides flexibility in building dynamic and structured user interfaces. Separate and modular templates support clean, consistent, and manageable interface development.

1.6.3 URLs

In Django, the URL routing system is a key component that connects user requests to the appropriate application processing logic. URL routing works by matching the URL requests received by the server with the URL patterns defined within the Django application. Each URL is mapped to a view that will handle the request and return an appropriate response, whether it's an HTML page, JSON data, or another type of response.

How URL Routing Works

URL routing in Django is configured through the `urls.py` file. Each Django application typically has a `urls.py` file that defines the URL patterns associated with that application. Django uses regex or path converters to match the URL patterns with incoming requests.

When a user accesses a URL in a Django application, the request is sent to Django's middleware, which then matches it with the defined URL patterns. If a match is found, the request is forwarded to the corresponding view. If no match is found, Django returns a 404 (Page Not Found) response.

Example URL Structure

The structure of the `urls.py` file in a Django application typically looks like this:

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

urlpatterns = [
    path('bots/', views.bot_list, name='bot_list'),
    path('bots/<int:id>/', views.bot_detail,
name='bot_detail'),
]
```

In the example above:

- The URL `bots/` is mapped to the view `bot_list`, which may be used to display a list of bots. We use `path()` to define the URL, which is more user-friendly compared to the regex approach in previous versions of Django.
- The URL `bots/<int:id>/` is mapped to the view `bot_detail`. `<int:id>` is a **path converter** that captures an integer from the URL and forwards it as an argument to the view. This allows us to create dynamic URLs that depend on specific data, such as the bot's ID.

Explanation of URL Patterns

The `path()` function accepts several arguments:

1. ***route***: This is the URL pattern to be matched. In the example above, `bots/` and `bots/<int:id>/` are the routes that Django will match.
2. ***view***: The view function that will be called if the route matches. For example, `views.bot_list` and `views.bot_detail`.
3. ***name***: A unique name for this URL pattern. This name can be used to dynamically create links in templates and views using the `reverse()` function or the `{% url %}` tag.

Connecting Application URLs to the Project

Once we have defined URLs within the application, we need to connect them to the main Django project through the `urls.py` file in the project directory. The project-level `urls.py` file serves as a link between the root URL of the application and the more specific URLs of each application.

For example, here is the `urls.py` file in the main project:

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

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

In this example, we use `include()` to incorporate the URLs from the **`bot_app`** into the project's URLs. Therefore, any request to `http://localhost:8000/bot_app/` will be forwarded to the `urls.py` file of **`bot_app`** and matched with the URL patterns defined there.

Dynamic URL Patterns

One of the advantages of Django's URL routing is the flexibility to match dynamic URL patterns. For example, we can capture parameters from the URL using converters, such as integer, slug, and others. Django also allows us to add custom converters if needed.

Example of using a path converter in a URL:

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

urlpatterns = [
    path('bots/<slug:slug>/',
views.bot_detail_by_slug, name='bot_detail_by_slug'),
]
```

In this example, we use the slug converter, which is suitable for more SEO-friendly URLs. The `bot_detail_by_slug` view will receive the slug value as an argument.

Using URL Names in Templates

By adding a name to the URL pattern, we can use it to dynamically create links in templates. For example, if we want to create a link to the bot detail page, we can use the `{% url %}` template tag:

```
<!-- bot_app/templates/bot_app/bot_list.html -->
<ul>
    {% for bot in bots %}
    <li>
        <a href="{% url 'bot_detail' bot.id
%}">{{ bot.name }}</a>
    </li>
    {% endfor %}
</ul>
```

The tag `{% url 'bot_detail' bot.id %}` will generate a link that corresponds to the URL defined in the `urls.py` file with the name `bot_detail`.

The URL routing system in Django provides a very flexible and easy way to manage routes within a web application. By using dynamic URL patterns, developers can easily handle various requests and forward them to the appropriate views. Django also allows the use of URL names to create more manageable and structured links, making it easier to maintain and develop applications in the future.

1.6.4 Views

Views are one of the most important components in the Django architecture. Essentially, views act as a bridge between application logic and what is displayed to users in their browsers. When a user makes an HTTP request to the application, Django processes it through URL routing, and then the request is forwarded to the relevant view. This view function is responsible for handling business logic and generating an appropriate response, which can be an HTML page, a JSON file, a text file, or even an image file.

How Views Work

Views in Django are typically written as functions or classes. A view function receives an HTTP request, performs various operations (such as retrieving data from a database or processing a form), and then produces a response that is displayed to the user. Views can also be configured to process different types of requests, such as GET and POST, to handle different actions on the same URL.

A simple example of a view function is as follows:

```python
# bot_app/views.py
from django.shortcuts import render
from .models import Bot

def bot_list(request):
    bots = Bot.objects.all()  # Retrieve all bot data 
from the database
    return render(request, 'bot_app/bot_list.html', 
{'bots': bots})
```

In the example above, the `bot_list` view receives a request from the user. First, we retrieve all bot objects from the database using `Bot.objects.all()`. Then, we use the `render()` function to render the `bot_list.html` template while sending the bot data to the template as context.

Comments:

- **`from django.shortcuts import render`**: We use `render()` to combine the template with the data that will be sent to the HTML page.

- **`Bot.objects.all()`**: This is a query to the Bot model that retrieves all bot objects from the database.

Types of Views

Django supports two types of views: **function-based views (FBVs)** and **class-based views (CBVs)**. Both serve the same purpose but with different approaches.

Function-Based Views (FBVs)

FBVs are a simpler, more straightforward approach where each view is a Python function. The example above is an FBV that executes business logic within a single function.

The main advantage of FBVs is their simplicity. However, as applications grow more complex and we have views that require similar behavior, using class-based views can be more effective because they support reusing common logic.

Class-Based Views (CBVs)

CBVs allow us to define views as classes. This approach is more modular and supports inheritance, making it suitable for reducing code duplication across various views. Django provides several built-in CBVs commonly used, such as `ListView`, `DetailView`, `CreateView`, and `UpdateView`.

For example, let's convert `bot_list` into a class-based view:

```
# bot_app/views.py
from django.views.generic import ListView
from .models import Bot

class BotListView(ListView):
    model = Bot
    template_name = 'bot_app/bot_list.html'
    context_object_name = 'bots'
```

In this CBV, we use the `ListView` class provided by Django to display a list of objects from the Bot model. We only need to specify the model, template, and context name for the template. Django automatically handles data retrieval and template rendering without requiring additional logic typically written manually in FBVs.

Using Views to Handle Various Request Types

One interesting feature of views in Django is the ability to handle both GET and POST requests within the same view. This is especially useful, for example, in form submissions, where GET is used to display the form, and POST is used to process the data submitted by the user.

The example below demonstrates how a view can handle both GET and POST requests simultaneously:

```
# bot_app/views.py
from django.shortcuts import render, redirect
from .forms import BotForm

def bot_create(request):
    if request.method == 'POST':
        form = BotForm(request.POST)
        if form.is_valid():
            form.save()  # Save the new bot to the
database
```

```
            return redirect('bot_list')  # Redirect
to the bot list page
    else:
        form = BotForm()
    return render(request, 'bot_app/bot_create.html',
{'form': form})
```

In this `bot_create` view, if the request method is POST, the form submitted by the user is processed. If valid, the data is saved to the database, and the user is redirected back to the bot list. If the request is GET, an empty form is displayed on the page for the user to fill out.

Separating Views for Different Purposes

It is important to separate view logic based on different actions. For example, a view used to display a list of objects may differ from a view used to create or edit an object. Django is very flexible in this regard, allowing us to build complex logic in a structured way.

The example below shows a common structure that separates views for different operations on bot objects:

```
# bot_app/views.py
def bot_detail(request, id):
    bot = Bot.objects.get(id=id)  # Retrieve bot data
based on ID
    return render(request, 'bot_app/bot_detail.html',
{'bot': bot})

def bot_update(request, id):
    bot = Bot.objects.get(id=id)
    if request.method == 'POST':
        form = BotForm(request.POST, instance=bot)
        if form.is_valid():
            form.save()
            return redirect('bot_detail', id=id)
    else:
        form = BotForm(instance=bot)
    return render(request, 'bot_app/bot_update.html',
{'form': form})
```

Here, `bot_detail` is used to display the details of a specific bot, while `bot_update` is used to edit a bot. Both separate logic to maintain code clarity and modularity.

Views in Django serve as the bridge between user requests and the results they will see. Through views, we can organize application logic and determine the responses sent back to users. With the capability to use function-based views and class-based views, Django offers flexibility in writing logic that suits the project’s needs. Views can also be configured to handle various request types, process forms, and efficiently manage user interactions with web applications.

1.6.5 Models

Models are one of the key components in Django used to handle interactions with the database. Django employs an Object-Relational Mapping (ORM) approach, allowing developers to interact with the database using Python objects instead of writing SQL commands directly. A model in Django acts as a representation of a table in the database, where each attribute of the model corresponds to a column in that table.

A model in Django is defined as a class that inherits from `django.db.models.Model`. Each attribute of this class will become a column in the database table. Django automatically creates the table based on the model we define when we run the database migration.

For example, let's see how we can define a `Bot` model for the `platform_bot` application:

```
# bot_app/models.py
from django.db import models

class Bot(models.Model):
    name = models.CharField(max_length=100)  # Column
for bot name
    description = models.TextField()  # Column for
bot description
    created_at =
models.DateTimeField(auto_now_add=True)  # Creation
time
    updated_at = models.DateTimeField(auto_now=True)
# Update time

    def __str__(self):
        return self.name  # Return bot name as string
representation
```

Comments:

- **`models.CharField(max_length=100)`**: Defines a column to store text with a maximum length of 100 characters.
- **`models.TextField()`**: Used to store longer text such as descriptions.
- **`models.DateTimeField(auto_now_add=True)`**: Stores the time when the object was first created.
- `models.DateTimeField(auto_now=True)`: Stores the time when the object was last updated.

- **`__str__`**: This method defines how the `Bot` object will be represented as a string when accessed in the admin interface or in the shell.

Using ORM to Interact with the Database

With the model defined, Django ORM allows us to perform database operations such as saving, updating, deleting, and retrieving data using Python objects. Django will translate these operations into the appropriate SQL queries behind the scenes.

For example, to add a new entry to the `Bot` table, we can use the `Bot` model as follows:

```
# Adding a new bot
bot = Bot(name="New Bot", description="Description of
the new bot")
bot.save()  # Save to database
```

In the code above, we create a `Bot` object with a specific name and description. After that, we call the `save()` method to save this object to the database.

Querying with Django ORM

Django's ORM not only allows us to add data but also to retrieve and manipulate data from the database in a very efficient manner. For instance, we can retrieve all bots in the database like this:

```
# Retrieving all bots
all_bots = Bot.objects.all()
```

For more specific queries, Django ORM provides various filter methods that can be used to retrieve data based on certain criteria:

```
# Retrieving the bot named 'New Bot'
bot = Bot.objects.get(name="New Bot")
```

The `get()` method will return a single object that meets the specified criteria or will raise an error if no object is found. If we want multiple objects that meet certain criteria, we can use `filter()`:

```
# Retrieving all bots whose names contain the word
'Bot'
bots = Bot.objects.filter(name__contains="Bot")
```

In the example above, `filter()` will return a `QuerySet` containing all bots whose names contain the word 'Bot'. Django ORM supports various types of queries that can be used to create complex and efficient filters.

Database Migration

Whenever we create or change a model, we need to run a database migration to reflect those changes in the table structure in the database. Django provides very easy migration management. After defining a model like `Bot`, we can run the migration command using the terminal:

```
$ python manage.py makemigrations bot_app
```

This command will generate a migration file based on the changes made to the model in the `bot_app` application. Next, we can apply the migration to the database with the command:

```
$ python manage.py migrate
```

With this command, Django will create the `Bot` table in the database according to the model definition.

Relationships Between Models

Django ORM also supports defining relationships between models, such as one-to-many, many-to-many, and one-to-one relationships. This allows us to create more complex data structures. For example, if we want each `Bot` to have one owner (user), we can add a `ForeignKey` relationship to the `Bot` model:

```
# bot_app/models.py
from django.contrib.auth.models import User

class Bot(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    owner = models.ForeignKey(User,
on_delete=models.CASCADE)  # Relation to User model
    created_at =
models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

By adding a `ForeignKey` to the `User` model, each bot is now associated with one owner. This relationship allows us to easily retrieve related information between two models.

Models in Django are a very important component because they allow us to interact with the database efficiently and intuitively. With robust ORM support, developers can write code that is easy to understand and maintain without worrying about the technical details of SQL. Models also allow us to manage database migrations automatically, keeping the synchronization between application models and the table structure in the database.

1.6.6 Forms

Forms in Django are an important component used to handle user input. In many web applications, forms are used to receive data from users, whether for registration processes, logging in, sending messages, or performing searches. Django provides a well-integrated form system to handle this input, including data validation, error management, and presentation of forms in the interface.

Django Forms are Python classes used to represent HTML forms and handle data validation. We can create forms manually or use Model Forms that automatically generate forms based on existing models.

Creating Forms with Django

There are two main ways to create forms in Django: manually by defining the form from scratch, or using ModelForm for forms based on models. First, let's see how to create a form manually.

For example, we want to create a form for users to add a new bot in the platform_bot application. We can define a form in the `forms.py` file as follows:

```
# bot_app/forms.py
from django import forms

class BotForm(forms.Form):
    name = forms.CharField(max_length=100)  # Input
for bot name
    description =
forms.CharField(widget=forms.Textarea)  # Input for
bot description
```

Comments:

- **`forms.CharField`**: This is the field to receive text input. `max_length=100` ensures the maximum length of input is 100 characters.
- **`widget=forms.Textarea`**: Replaces the standard text input with a textarea, which is more suitable for receiving long text such as descriptions.

This form can be displayed in a template and linked to a view to handle user input.

Using ModelForm

If our form is model-based, it is more efficient to use ModelForm, where Django will automatically generate the form based on the model we created. For example, we can create a form for the Bot model like this:

```
# bot_app/forms.py
from django import forms
from .models import Bot

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot  # Model used as the basis for
the form
        fields = ['name', 'description']  # Fields we
want to display in the form
```

Comments:

- **`model = Bot`**: Specifies that this form is based on the Bot model.
- **`fields = ['name', 'description']`**: Indicates the fields we want to include in the form. In this case, we are only using `name` and `description`.

With ModelForm, Django will automatically generate the appropriate HTML form that corresponds to the model and fields we specified, and handle basic validation such as text length limits.

Displaying Forms in Templates

After the form is defined, the next step is to display it in a template. We can include this form in an HTML template by utilizing Django's template tags. For example, we want to display the form for adding a bot in the `add_bot.html` page:

```
<!-- templates/add_bot.html -->
<form method="post">
  {% csrf_token %}
  {{ form.as_p }}  <!-- Display the form in <p>
elements -->
  <button type="submit">Save</button>
</form>
```

Comments:

- **`{% csrf_token %}`**: Used to protect the form from CSRF (Cross-Site Request Forgery) attacks.
- **`{{ form.as_p }}`**: This template tag renders the form as HTML paragraphs (`<p>`), meaning each field will be surrounded by a `<p>` tag. Django also provides other methods like `{{ form.as_table }}` and `{{ form.as_ul }}` to render the form as a table or unordered list.

Handling Form Input in Views

Once the form is submitted, Django requires logic to handle the incoming data. This logic is typically defined in views. Here's an example of how we can process the `BotForm` in a view:

```
# bot_app/views.py
from django.shortcuts import render, redirect
from .forms import BotForm

def add_bot(request):
    if request.method == 'POST':  # If the form is
submitted
        form = BotForm(request.POST)
        if form.is_valid():  # Check data validation
            # Handle the validated form data
            # In this case, we simply print the data
to the terminal
            print(form.cleaned_data)
            return redirect('manage_bots')  #
Redirect the user after successful submission
    else:
        form = BotForm()  # If the request is GET,
display an empty form

    return render(request, 'add_bot.html', {'form':
form})  # Render the template with the form
```

Comments:

- **`request.method == 'POST'`**: Checks whether the form has been submitted.
- **`form.is_valid()`**: Checks if the submitted data meets all the form validation rules.
- **`form.cleaned_data`**: Contains the validated data, ready to use.
- **`redirect('manage_bots')`**: Redirects the user to another page after the form has been successfully processed, in this case, to the manage_bots page.

Form Validation

Django Forms provide automatic validation mechanisms. For example, if we set `max_length=100` on the `name` field, Django will automatically raise an error if the user enters more than 100 characters. However, we can also add custom validation as needed by writing specific methods in the form.

For example, if we want to ensure that the bot name must be unique, we can add validation like this:

```
# bot_app/forms.py
from django import forms
from .models import Bot

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot
        fields = ['name', 'description']

    def clean_name(self):
        name = self.cleaned_data.get('name')
        if Bot.objects.filter(name=name).exists():
            raise forms.ValidationError("The bot name
is already in use.")
        return name
```

In the code above, the `clean_name` method will be called during the validation process. If the bot name already exists in the database, the form will raise an error, preventing the user from entering a duplicate name.

Django Forms are a powerful and flexible tool for handling user input. With robust validation systems and tight integration with models through ModelForm, forms in Django become easier to manage and process. This makes handling user input safer and more efficient, saving development time with many built-in features that can be directly utilized without needing to write complex manual logic.

1.6.7 Static Files and Media Files

In web application development, we often need to handle static files such as CSS, JavaScript, or images, as well as media files uploaded by users, such as profile pictures, videos, or other documents. Django provides a very flexible and structured mechanism for managing both types of files: Static Files and Media Files.

Static Files

Static files are files that do not change while the application is running, such as CSS files, JavaScript files, and images used for the user interface. Django has a built-in framework for handling static files, which makes it easy to store, organize, and serve static files in the application.

To handle static files, we first need to define the directory where these files will be stored. Generally, we define a `static/` folder in each application. We also need to add configuration in the `settings.py` file to tell Django where to look for static files.

The configuration in `settings.py` for static files is as follows:

```
# settings.py
STATIC_URL = '/static/'  # Base URL for static files
STATICFILES_DIRS = [
    BASE_DIR / 'static',  # Directory where we store
static files
]
```

Comments:

- ***STATIC_URL***: The base URL where static files can be accessed. Typically `/static/`, meaning if there is a CSS file at `static/css/style.css`, it can be accessed via the URL `/static/css/style.css`.
- ***STATICFILES_DIRS***: A list of additional directories where Django will look for static files besides each application.

After this is set up, we can store CSS files, JavaScript files, or images in the `static/` folder in a specific project or application. For example, in the `bot_app` application, we could have the following directory structure:

```
bot_app/
    static/
        bot_app/
            css/
            js/
            img/
```

CSS or JavaScript files will be loaded in the template using the template tag `{% static %}`. For example, to load the `style.css` file in a template, we could write:

```
<!-- templates/base.html -->
<link rel="stylesheet" type="text/css" href="{%
static 'bot_app/css/style.css' %}">
```

Media Files

Media files are files uploaded by users, such as profile pictures, documents, videos, or other files. Django also provides an easy way to handle these media files using `MEDIA_URL` and `MEDIA_ROOT`.

In `settings.py`, we need to define where media files will be stored in the file system (in the `MEDIA_ROOT` variable) and how they will be accessed via URLs (in the `MEDIA_URL` variable).

```
# settings.py
MEDIA_URL = '/media/'  # Base URL for media files
MEDIA_ROOT = BASE_DIR / 'media'  # Directory where
media files will be stored
```

Comments:

- ***MEDIA_URL***: The base URL for accessing user-uploaded media files. For example, if an image file is uploaded and stored at `media/images/avatar.jpg`, it can be accessed via the URL `/media/images/avatar.jpg`.
- ***MEDIA_ROOT***: The path where all media files will be stored in the file system. We usually store these files in a `media/` folder in the project.

To store media files in the application, we need to configure our models to support file uploads. For example, if we want users to upload an avatar image when they register a bot in our application, we can add an image field in the `Bot` model like this:

```
# bot_app/models.py
from django.db import models
```

```
class Bot(models.Model):
    name = models.CharField(max_length=100)  # Bot
name
    description = models.TextField()  # Bot
description
    avatar = models.ImageField(upload_to='avatars/')
# Field for storing avatar image
```

Comments:

- ***ImageField***: This field is used to store image files in the database. The argument `upload_to='avatars/'` indicates that all uploaded image files will be stored in the `media/avatars/` folder.

To display and access media files in the template, we can use the URL generated by `MEDIA_URL`:

```
<!-- templates/bot_detail.html -->
<img src="{{ bot.avatar.url }}" alt="Bot Avatar">
```

Comments:

- ***bot.avatar.url***: This generates the complete URL of the uploaded image file to be displayed on the page.

Serving Static and Media Files in Development

In a development environment, Django does not automatically serve static and media files. We need to add a little configuration in `urls.py` to allow these files to be accessed during development.

In `urls.py`, add the following:

```
# project/urls.py
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # Other URL patterns
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)  # Serving media
files
    urlpatterns += static(settings.STATIC_URL,
document_root=settings.STATICFILES_DIRS[0])  #
Serving static files
```

Comments:

- ***static()***: This function is used to serve static and media files in a development environment. In production, we would use a web server like Nginx or Apache to serve these files.

With the built-in support for static files and media files, Django makes file management in web applications much easier. All static files can be well-managed within the `static/` directory, and user media files can be stored and accessed using the `MEDIA_URL` and `MEDIA_ROOT` mechanisms. This provides great flexibility in developing interactive interfaces while ensuring user files can be accessed and processed securely.

1.6.8 Middleware

In Django's architecture, **middleware** is an essential component that operates between the request and response. Simply put, middleware is a layer that allows us to process data from the request before it reaches the view, or modify the response before it is sent back to the client.

Middleware provides additional control over how HTTP requests are processed, giving us the opportunity to implement various functions such as authentication, logging, response modification, or even blocking certain requests.

Basic Functions of Middleware:

Middleware in Django functions as a hook that is executed sequentially every time an HTTP request is received by the server. Each middleware can access the request before it reaches the view and also the response after the view has been processed. This allows us to insert logic that runs either at the early stage (for example, authentication) or the late stage (for example, setting specific headers).

In the normal workflow process, Django does the following:

1. Receives a request from the client.
2. Passes the request through the various middleware we have defined.
3. Sends the processed request to the relevant view.

4. After the view generates a response, the response is processed through middleware again before being sent to the client.

This flow makes middleware very flexible and useful in various scenarios.

Example of Middleware Usage:

For example, if we want to create a simple middleware that logs every incoming HTTP request to our application, we can create a custom middleware like this:

```python
# project/middleware.py
import logging

logger = logging.getLogger(__name__)

class LogRequestMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Log request method and path
        logger.info(f"Request Method:
{request.method}, Path: {request.path}")

        # Proceed to the next view
        response = self.get_response(request)

        # Return the processed response
        return response
```

Comments:

- This middleware is created in the `middleware.py` file located in the project folder.

- **`__call__()`**: This is the main method executed every time there is an HTTP request. Here we can add additional logic, such as logging information about the incoming request.
- `self.get_response(request)` continues the request to the next middleware or directly to the view.

After creating this middleware, we need to register it in the `settings.py` file so that Django recognizes and executes it:

```
# settings.py
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware
',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddlew
are',

'django.contrib.messages.middleware.MessageMiddleware
',

'django.middleware.clickjacking.XFrameOptionsMiddlewa
re',

    # Our custom middleware
    'project.middleware.LogRequestMiddleware',
]
```

Comments:

- In `settings.py`, our middleware is added to the `MIDDLEWARE` sequence, so Django will run it alongside other built-in middleware.

Types of Built-in Django Middleware:

Django also provides a number of built-in middleware that are very useful in managing various aspects of requests and responses. Some commonly used middleware include:

1. ***SecurityMiddleware***: Applies security practices such as Strict-Transport-Security to ensure all requests use HTTPS.
2. ***SessionMiddleware***: Provides support for session management, allowing us to track users as they interact with the application.
3. ***CsrfViewMiddleware***: Enables protection against CSRF (Cross-Site Request Forgery) for any request involving a form.
4. ***AuthenticationMiddleware***: Manages user authentication by associating authenticated users with each request.

With these built-in middleware, Django provides many security features and conveniences in handling requests without requiring us to write additional code.

Benefits of Using Middleware:

Some benefits of using middleware in Django are:

- ***Modularity***: Middleware allows us to separate different logic into manageable components. For example, we can place all authentication-related logic in one middleware and all logging-related logic in another middleware.
- ***Flexibility***: With middleware, we can make various modifications to requests and responses without disrupting views or the main application logic. This is

very useful for adding security features, logging, caching, or header manipulation.

- ***Request Management***: Middleware gives us full control over the request flow, allowing us to block or modify requests before they reach the view or manipulate the generated response.

Middleware is a crucial tool in Django's architecture, enabling us to handle various application needs without disrupting core logic. Whether we want to add authentication, perform logging, or protect the application from CSRF attacks, middleware provides an elegant solution that integrates well within the Django framework.

1.6.9 Signals

Signals in Django are a feature that allows us to implement event-driven programming. By using signals, we can make application components react automatically to specific events that occur within the system. This is especially useful when we want to execute additional code in response to changes in models, user actions, or other events in the application without disrupting the main logic.

Concept of Signals

Signals in Django serve to connect events with responses. When an event occurs, a signal will "send" a notification, and the functions connected to that signal will automatically run. This allows us to separate event-driven logic from the main application code, creating a more modular and manageable system.

Some commonly used signals in Django include:

- ***pre_save*** and ***post_save***: Triggered before and after a model is saved to the database.
- ***pre_delete*** and ***post_delete***: Triggered before and after a model is deleted from the database.
- ***request_started*** and ***request_finished***: Triggered at the start and end of the request cycle.
- ***user_logged_in*** and ***user_logged_out***: Triggered when a user logs in or out.

Creating and Connecting Signals

To understand how signals work, let's create a simple signal that sends a notification whenever a new User object is created. We will use the `post_save` signal to execute a function after the User model is successfully saved.

```
# apps/users/signals.py
from django.db.models.signals import post_save
from django.contrib.auth.models import User
from django.dispatch import receiver

# This function will be called after a user is
successfully saved
@receiver(post_save, sender=User)
def send_welcome_email(sender, instance, created,
**kwargs):
    if created:
        # Send a welcome email to the new user
        print(f"Welcome {instance.username}, thank
you for signing up.")
```

Comments:

- This file is placed in the `signals.py` folder within the `users` application.
- The `send_welcome_email` function will run every time a new User object is saved to the database.
- `@receiver(post_save, sender=User)` indicates that this signal only applies to the User model and will be triggered after the save process.

To ensure this signal is active, we need to register it. Typically, this is done in the `apps.py` file of the application:

```
# apps/users/apps.py
from django.apps import AppConfig

class UsersConfig(AppConfig):
    name = 'users'

    def ready(self):
        # Registering the signal when the application
is ready
        import users.signals
```

Comments:

- The `apps.py` file is modified to ensure that the signal is loaded when the `users` application is initialized by Django.
- The `ready()` method is a special method called when the application is loaded, and this is where we import the `signals.py` file.

Built-in Django Signals

Django provides various built-in signals that are useful for many situations. Some examples of commonly used signals in application development include:

1. ***pre_save***: Called before an object is saved. This signal is useful when we want to modify data before it is written to the database.
2. ***post_save***: Called after an object is saved. This signal is suitable for executing additional tasks, such as sending notifications or updating related data.
3. ***pre_delete***: Called before an object is deleted. We can use this signal to clean up related data before the deletion occurs.
4. ***post_delete***: Called after an object is deleted, allowing us to follow up on the deletion event, such as deleting related files.
5. ***request_started*** and ***request_finished***: Useful for logging or monitoring the lifecycle of an HTTP request.

Example of Using a Signal for Data Cleanup

For example, we have a `Profile` model that is related to the `User`. If a user is deleted, we want to automatically delete the related profile. We can use the `pre_delete` signal for this purpose.

```python
# apps/users/signals.py
from django.db.models.signals import pre_delete
from django.contrib.auth.models import User
from django.dispatch import receiver
from .models import Profile

@receiver(pre_delete, sender=User)
def delete_user_profile(sender, instance, **kwargs):
    # Delete user profile before user is deleted
    Profile.objects.filter(user=instance).delete()
```

Comments:

- Here, we use the `pre_delete` signal to ensure that when a User object is deleted, the related profile is also removed from the database.
- `Profile.objects.filter(user=instance).delete()` deletes all Profile objects related to the user being deleted.

Benefits of Using Signals

Signals provide several important benefits in Django's architecture:

1. ***Modular Programming***: Signals allow us to separate event-driven logic from the main application logic. This creates a cleaner and more modular system, making it easier to manage and debug.
2. ***Reduction of Boilerplate Code***: By using signals, we can reduce repetitive code. For instance, if we need to execute a specific action each time a model is saved, we just define the relevant signal rather than writing that code in multiple places.
3. ***Automatic Handling***: Signals are well-suited for automated tasks, such as sending welcome emails, updating related data, or logging events.

Signals are a powerful tool in Django, enabling us to add event-driven logic without disrupting the main application structure. With signals, we can handle various events, such as model saving or deletion, automatically and efficiently. The use of signals enhances the modular and flexible nature of Django, giving developers more control over how events are processed within the application.

1.7 Why Choose Django?

Django is one of the most popular and widely used web frameworks globally. There are several compelling reasons why many developers choose Django as the foundation for building their web applications. These reasons are closely related to the various features Django offers and its ability to simplify the development of complex web applications. In this section, we will explain the advantages of Django and why it has become a top choice for many web developers.

Fast and Efficient

One of Django's key strengths is the speed it offers during development. Django is designed to help developers turn ideas into finished products quickly and efficiently. This is because Django provides many tools and features that simplify web application creation. From an automatic admin interface to a sophisticated ORM, Django offers everything developers need to build web applications in significantly less time than starting from scratch.

In the terminal, we can create a new Django project very quickly. For example, to create a new project, we simply run the following command:

```
$ django-admin startproject platform_bot
```

Comment:

- This command creates the basic framework of a new Django project in the specified directory. Django automatically sets up the folder structure, configuration files, and other essential components.

With the project structure ready, we can start developing without worrying about a complicated initial setup.

Integrated Security

Security is one of the top priorities in web application development. Django has integrated security features, including protection against various types of attacks like **SQL injection**, **cross-site scripting (XSS)**, **cross-site request forgery (CSRF)**, and **clickjacking**. These features allow developers to focus on building application functionality without worrying about basic security.

For example, Django automatically protects applications from CSRF attacks through the use of a CSRF token in forms that receive user input. Django automatically inserts this token in forms, so we don't need to add it manually.

```
<form method="POST">
    {% csrf_token %}
    <!-- Other form fields -->
</form>
```

Comment:

- In the HTML template, we only need to add **`{% csrf_token %}`** inside the form to ensure that the form is protected from CSRF attacks.

Easy Scalability

Django is designed with scalability in mind. This means that applications built with Django can easily be optimized to handle increasing user traffic loads. Whether for small startups or large enterprise applications, Django can easily handle various project scales. Django supports multiple databases and can easily be configured to support caching, load balancing, and other distributed systems required as the application grows.

Strong Community and Documentation

One of the most beneficial aspects of Django is its large and active user community. This community provides many resources, such as tutorials, discussion forums, and open-source libraries that can be used to add functionality to applications easily. Additionally, Django’s documentation is very comprehensive and easy to understand, even for developers new to the framework.

If we encounter issues during development, chances are the solution is already available in Django’s official documentation or through the active community on platforms like Stack Overflow and GitHub.

Rich Ecosystem and Built-in Features

Django not only provides tools for basic development but also includes various built-in features that greatly assist developers. Features like authentication, URL routing, form handling, and an automatic admin interface make Django a very complete framework.

Moreover, Django has additional libraries that can be easily integrated to add specific functionalities, such as Django REST Framework for building APIs.

To set up URL routing, for example, we only need to define URL patterns in the **`urls.py`** file:

```
# apps/platform_bot/urls.py
from django.urls import path
from . import views

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

Comment:

- In this `urls.py` file, we define the URLs that will map requests to the `home` and `about` functions in `views.py`. This way, Django simplifies the routing process and connects URLs with views.

Django Offers “Batteries Included”

One of Django’s core philosophies is “batteries included,” which means Django comes with many tools and features built into the framework, so developers don’t need to seek external solutions for most development needs. For example, Django includes a built-in ORM to manage database interactions, form handling to simplify user input validation and processing, and an automatic admin interface that is incredibly useful for managing application data without writing additional code.

With Django, many development aspects are already automated or provided by default, saving developers time and effort in setting up basic web application features.

Django is a powerful, flexible, and efficient framework for web development. Its speed, security, scalability, and large community make Django an attractive choice for many developers. With numerous built-in features that simplify the development process, Django allows developers to focus on building the business logic and unique functionality of their applications instead of getting stuck on basic technicalities.

1.7.1 The Advantages of Django in Web Development

Django, a Python-based web framework, is renowned for its ability to simplify the development of complex web applications. Its advantages stem from the wide array of features and functionalities it offers. In this section, we will delve deeper into how Django streamlines the web application development process and why it has become a popular choice among developers.

Organized Project Structure

One of Django’s key strengths is its well-organized project structure. When starting a new Django project, the framework automatically generates the necessary folders and files. This structure includes directories for apps, configuration files, and other essential components, making the development process more structured.

For example, when creating a new project using the following command:

```
$ django-admin startproject platform_bot
```

Comment:

- This command generates the main project folder **`platform_bot`** with sub-folders containing **`manage.py`**, **`platform_bot/settings.py`**, **`platform_bot/urls.py`**, and **`platform_bot/wsgi.py`**. This structure makes code management and organization more straightforward.

With this setup, developers can quickly understand and work on different parts of the application without spending time on setting up the basic project framework.

Automatic Admin Interface

Django provides an outstanding feature called the automatic admin interface. This allows developers to manage application data directly through a web interface, which is automatically generated. The admin interface is particularly useful for administrative tasks such as adding, modifying, and deleting data from application models.

To use the admin interface, we only need to register our models in **`admin.py`**. For example, if we have a **`Bot`** model, we can register it like this:

```
# apps/platform_bot/admin.py
from django.contrib import admin
from .models import Bot

admin.site.register(Bot)
```

Comment:

- By registering the **`Bot`** model in the admin site, Django automatically generates an administrative interface that allows users to manage **`Bot`** entities through the web browser.

This feature saves developers a significant amount of time since there's no need to build an administrative interface from scratch. With minimal configuration, we gain access to a highly functional tool.

Powerful Object-Relational Mapping (ORM)

Django provides a robust ORM system to handle database interactions. The ORM allows developers to interact with the database using Python objects instead of writing raw SQL queries. This not only simplifies code maintenance but also enhances application security by preventing SQL injection attacks.

For instance, to retrieve all **`Bot`** entities from the database, we can use the following code:

```
# apps/platform_bot/views.py
from django.shortcuts import render
from .models import Bot

def bot_list(request):
    bots = Bot.objects.all()  # Retrieves all Bot 
entities from the database
    return render(request, 'bot_list.html', {'bots': 
bots})
```

Comment:

- Using **`Bot.objects.all()`**, we can fetch all **`Bot`** entities without writing manual SQL queries, making data retrieval more straightforward and the code cleaner.

Simple Form Handling

Django also offers powerful tools for handling forms and validating user input. With **`forms.py`**, we can define forms, validations, and input processing logic in a structured way.

For example, here's a form to create a `Bot` entity:

```
# apps/platform_bot/forms.py
from django import forms
from .models import Bot
```

```
class BotForm(forms.ModelForm):
    class Meta:
        model = Bot
        fields = ['name', 'description']  # Fields to
be displayed in the form
```

Comment:

- Using **`ModelForm`**, we can automatically generate a form from the **`Bot`** model, including appropriate validation for the fields defined.

Flexible Routing System

Django's routing system is both flexible and powerful, enabling us to define URL patterns and link them to views. This system simplifies how the application handles different URL requests and makes developing applications with many endpoints easier.

For instance, to define URL patterns in **`urls.py`**:

```
# apps/platform_bot/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='home'),
    path('bot/<int:id>/', views.bot_detail,
name='bot_detail'),
]
```

Comment:

- By defining URL patterns in **`urls.py`**, we can link specific URLs to the **`home`** and **`bot_detail`** functions in **`views.py`**, simplifying the handling of requests and URL routing.

Django simplifies the development of complex web applications through its organized project structure, automatic admin interface, powerful ORM, efficient form handling, and flexible routing system. These features allow developers to focus on application logic and functionality while Django handles the technical complexities. These advantages make Django an appealing choice for building scalable and well-managed web applications.

1.7.2 *Django vs Other Frameworks*

When choosing a framework for web development, many factors must be considered, including features, ease of use, and the available ecosystem. Django is one of the popular web frameworks, but many developers also consider other frameworks like Flask, Ruby on Rails, and Laravel. In this subsection, we will compare Django with these frameworks, highlighting the strengths and weaknesses of each.

Django vs Flask
Flask is a Python-based web framework often compared with Django.

Here are some key differences between them:
Approach and Features:

- ***Django:*** Django is a "batteries-included" framework, meaning it provides many built-in features like an admin interface, ORM, and a templating system. This allows developers to start quickly without needing much additional configuration.

- ***Flask:*** Flask, on the other hand, is a micro-framework. This means Flask provides a very lightweight foundation and lets developers choose and integrate various additional components as needed. Flask offers more flexibility in selecting tools but requires more manual configuration.

Advantages of Django:

- ***Built-in Features:*** With features like the automatic admin interface and ORM, Django speeds up application development by providing ready-to-use tools.
- ***Project Structure:*** Django provides a well-organized project structure, making it easier to manage code in larger applications.

Advantages of Flask:

- ***Freedom of Choice:*** Flask gives developers the freedom to choose the libraries and tools they want, suitable for projects with specific needs.
- ***Lightweight and Simple:*** Flask is simpler and more lightweight, making it ideal for small applications or microservices that don't need many built-in features.

Django vs Ruby on Rails

Ruby on Rails (RoR) is a popular web framework based on Ruby.

Here is a comparison between Django and RoR:

Approach and Features:

- ***Django:*** Django uses the MVT (Model-View-Template) approach and offers many built-in features to accelerate development. Django is also known for the "Don't Repeat Yourself" (DRY) principle, which encourages efficient and structured coding.
- ***Ruby on Rails:*** RoR also uses the DRY principle and "Convention over Configuration," reducing the need for configuration by following certain conventions. RoR provides many automated features that make web development easier.

Advantages of Django:

- ***Project Management:*** Django's project structure makes it easier to manage and organize code.
- ***Security:*** Django is known for its strong security features, including protection against SQL injection, XSS, and CSRF attacks.

Advantages of Ruby on Rails:

- ***Ease of Use:*** RoR is famous for its ease of use and the ability to build web applications quickly, thanks to established conventions.
- ***Ecosystem and Community:*** RoR has a rich gem ecosystem, offering many libraries and tools to speed up development.

Django vs Laravel

Laravel is a popular web framework based on PHP.

Here’s a comparison between Django and Laravel:

Approach and Features:

- ***Django:*** As mentioned earlier, Django offers many built-in features and uses Python, a language known for its clean and easy-to-read syntax.
- ***Laravel:*** Laravel also offers many built-in features and uses PHP. Laravel includes features like Eloquent ORM, Blade templating, and Laravel Mix for front-end asset management.

Advantages of Django:

- ***Code Quality:*** Django promotes code quality by following strong design principles and providing features to keep code clean and well-structured.
- ***Python Ecosystem:*** Django leverages Python's vast ecosystem, allowing integration with many other Python tools and libraries.

Advantages of Laravel:

- ***Modern Features:*** Laravel offers modern features such as advanced routing, middleware, and a strong database migration system.
- ***Blade Templating:*** Blade, Laravel's templating system, provides a simple and easy-to-use syntax for creating dynamic views.

Choosing between Django, Flask, Ruby on Rails, and Laravel depends on the specific needs of the project and the developer's preferences. Django offers many built-in features and a well-organized project structure, making it ideal for large web applications with complex needs. Flask provides flexibility and simplicity, perfect for smaller applications or microservices. Ruby on Rails is known for its ease of use and conventions that simplify web application development, while Laravel offers modern features and a strong PHP ecosystem. Each framework has its strengths and weaknesses, and the final decision should be based on the project's needs and the development team's expertise.

Django, with its advantages, is a very strong choice for developers looking to build sophisticated, secure, and scalable web applications. Although other frameworks may offer advantages in certain situations, Django provides an ideal balance between features, ease of use, and wide community support.

1.7.3 Django for Small to Large-Scale Projects

Django is an incredibly flexible framework that can be used for a wide variety of projects, ranging from small web applications to large systems with complex requirements. This versatility makes it a popular choice among developers, whether working on small-scale or large-scale projects.

Using Django in Small Projects

For small projects, Django offers a lot of convenience. With its automatically generated project structure and built-in features like the admin interface and ORM, developers can quickly build simple web applications. Some of the advantages of Django for small projects include:

- ***Simplicity***: Django simplifies web application development by offering many built-in features that speed up the process.
- ***Built-in Admin Interface***: Django's admin interface allows easy data management without requiring extensive coding.
- ***ORM (Object-Relational Mapping)***: Django's ORM makes database interaction simple, allowing for easy database queries without needing direct SQL.

For example, if you want to build a simple blog or information application, Django enables you to set up models, views, and templates quickly with minimal configuration. To create a simple blog application, only a few commands are needed to set up the project and the basic model.

```
$ django-admin startproject myblog
$ cd myblog
$ python manage.py startapp blog
```

Then, you can quickly create models and views in the `models.py` and `views.py` files within the blog app, set up URLs, and create templates to display the blog content.

Django for Large-Scale Projects

Django is also well-suited for large projects with complex requirements. Django's features and architecture are designed to support the development of large systems with various features, including:

- ***Modularity***: Django supports an app-based architecture, allowing developers to break a large project into several separate apps. This makes managing and developing the application easier.
- ***Scalability***: Django is built with performance and scalability in mind. Features such as caching, query optimization, and the ability to handle a high number of requests concurrently ensure that applications can grow as user demand increases.
- ***Security***: Django includes many built-in security features that protect applications from various attacks, such as CSRF, XSS, and SQL Injection, which is crucial for large applications handling sensitive data.

For large projects such as e-commerce platforms or content management systems, Django offers tools and features to manage complexity:

1. ***Custom Middleware and Signals***: To handle complex processes and event-driven tasks.
2. ***Advanced ORM***: To perform complex database queries and manage relationships between models.
3. ***Caching and Load Balancing***: To improve performance and handle high traffic.

For example, when building a large e-commerce platform, Django allows you to use different apps to manage functions like product management, payment processing, and user management, keeping each app separate but well-integrated.

Django is an extremely flexible framework that can adapt to various project scales. By providing powerful tools and features, Django enables developers to quickly get started on small projects and manage the complexity of larger applications. Django's flexibility and robustness ensure that applications can grow as user needs and application complexity increase.

With the steps outlined, you can effectively leverage Django for projects of any size, starting quickly with small projects and managing the complexity of larger applications.

1.7.4 Real-World Use Cases of Django

Django, as a powerful and flexible web framework, has been used by many real-world applications and companies. Below are several case studies illustrating how various organizations leverage Django to meet their needs.

Instagram

Instagram is one of the most famous examples of a large-scale application that uses Django. Initially, Instagram was built as a Django-based app, utilizing the framework’s strengths to handle large scale and high data volumes. Django provides Instagram with the ability to:

- ***Handle High Traffic***: With its caching features and optimized performance, Django helps Instagram manage millions of active users and photos uploaded daily.
- ***Scalability***: Django allows Instagram to easily scale, adding new features like Stories and IGTV without sacrificing performance.

- ***Security***: Django offers various security features that protect user data and prevent common attacks, which is critical for a social media platform.

Pinterest

Pinterest, a platform for sharing images and ideas, also uses Django in its architecture. Pinterest leverages Django to:

- ***Content Management***: Django helps Pinterest manage and present visual content on a large scale with high efficiency.
- ***User Management***: With Django's authentication and authorization features, Pinterest can securely manage user data and preferences.
- ***Performance Optimization***: Pinterest uses Django to serve content quickly and efficiently to users, even under heavy traffic conditions.

Disqus

Disqus, a commenting platform used by various websites, is built with Django. Some of the benefits Disqus gains from Django include:

- ***Comment Management***: Django simplifies managing user comments across different websites.
- ***Easy Integration***: Django allows Disqus to integrate additional features and adapt to the specific needs of various websites.
- ***Security and Scalability***: Django provides protection against attacks and ensures the system can handle a large volume of comments.

The Washington Post

The Washington Post, one of the largest newspapers in the United States, uses Django for some parts of their website. Django provides:

- ***Content Management for News***: Django simplifies the management and presentation of news articles efficiently.
- ***Design Flexibility***: With Django, The Washington Post can customize design and layout to meet editorial and technical needs.
- ***Data Management***: Django helps manage user data and their interactions with the news content.

NASA

NASA uses Django for various internal and public-facing applications. With Django, NASA is able to:

- ***Data and Application Management***: Django allows NASA to manage research data and applications with high accuracy and efficiency.
- ***Facilitate Collaboration***: Django supports collaboration tools and project management, helping NASA teams work together effectively.
- ***Handle Large Data Volumes***: Django's capability to handle large data allows NASA to process and present complex information.

Django has proven its capability to be used in a wide range of applications and organizations, from small startups to large enterprises and government agencies. Its ability to handle large data volumes, scalability, security, and ease of integration make it an excellent choice for many types of applications. The case studies above demonstrate how Django can be applied to meet various real-world needs and challenges.

By harnessing the power of Django, organizations can build strong and flexible applications while efficiently managing complexity and growth.

Chapter Conclusion

In this chapter, we have explored the fundamentals of Django, a highly powerful and efficient web framework. We began by understanding what Django is and why it is so popular among web developers. Django offers several advantages, including ease of developing fully functional web applications and strong out-of-the-box features.

We also discussed the Model-View-Controller (MVC) architecture in the context of Django, known as Model-View-Template (MVT). Understanding the key components of Django—such as apps, templates, URLs, views, models, forms, and static and media files—is crucial for developing well-structured and easily manageable web applications.

Key Components:

- ***App***: Modular units that make it easier to develop different parts of a web application.
- ***Template***: A templating system that enables the creation of dynamic user interfaces.
- ***URLs***: URL configurations that link user requests to the appropriate views.
- ***Views***: Functions or classes that process requests and generate responses.
- ***Models***: Definitions of data structures that interact with the database.
- ***Forms***: Tools for handling user input and data validation.

- ***Static and Media Files***: Management of static and media files necessary for the user interface and uploaded content.

Understanding these components will simplify the process of developing your web application. With this strong foundation, you are ready to move on to the practical steps in the following chapters, where we will begin building Django applications and applying what you have learned to create real-world and functional web solutions.

In the next chapter, we will start by setting up a Django project, step by step, focusing on applying theory into practice. Don’t worry if some concepts seem complex; the practical steps will help clarify how Django works in depth.

With a spirit of learning and exploration, let’s move forward to the next chapter and start building our Django web applications!

Chapter 2 - Preparation and Installation of the Development Environment

In this chapter, we will begin our journey with the ***platform_bot*** project, a bot application that will run using Django and integrate with various platforms such as Telegram and WhatsApp. The main goal of this chapter is to prepare the development environment required to efficiently develop and run this bot.

2.1 Chapter Objective

We will start by setting up the operating system and ensuring that the necessary software is available and ready to use. This includes installing Python, which is the primary programming language we will be using, as well as setting up a virtual environment that will help us keep this project separate from others and ensure all required dependencies are available.

Next, we will install Django, the web framework that will serve as the foundation of our bot application. Django will make it easier for us to build a robust and scalable web application. After that, we will install and configure ***Ngrok***, a very useful tool that will connect our local application to the outside world via webhooks. Ngrok will enable our bot to receive and respond to messages from platforms such as Telegram and WhatsApp in real-time, even when the application is running on a local server.

Additionally, we will choose and install the appropriate bot framework for integration with the targeted platforms. This will include installing additional libraries and configuring the basic settings to ensure that the bot can communicate properly through the webhook.

Finally, once all installations are complete, we will perform verification to ensure that every component of our development environment is functioning as it should. This includes running the Django development server, checking the Ngrok connection, and making sure that the bot can interact with the chosen platforms.

By completing this chapter, you will have a ready-to-use development environment to continue developing the bot in the following chapters. The steps we take here will provide a solid foundation for building advanced bot features and functionalities in the future.

This section provides a clear and detailed overview of what will be achieved in this chapter, along with the steps that will be taken to set up the development environment for the ***platform_bot*** project.

2.2 Preparation Of The Environment

Before proceeding with software installation, it is important to select the operating system that will be used for developing the ***platform_bot*** project. The operating system you choose can affect how you set up your development environment, as well as how you manage and run the application.

2.2.1 Choosing an Operating System

Windows is a commonly used operating system and often the first choice for many developers. If you are using Windows, ensure that you have tools like PowerShell or Command Prompt to run terminal commands. It is also recommended to use Windows Subsystem for Linux (WSL) if you want an environment closer to Linux.

macOS is an excellent option if you prefer working in a Unix-like environment. macOS comes with many development tools pre-installed, and you can use the Terminal to execute commands. macOS users often find that development tools are readily available and easily accessible.

Linux is very popular among developers due to its flexibility and power. If you're using Linux, you can use the terminal to install and configure the necessary software with efficient commands. Linux is often the primary choice for servers and production environments because of its customizability and stability.

Preparation and Installation of the Development Environment

Regardless of the operating system you choose, the basic steps for preparing the development environment are similar. You need to ensure that your system has access to the internet to download the required software and that you have sufficient permissions to install and configure the necessary tools.

For ***Windows***, make sure to install Python from the official website and consider using a package manager like ***Chocolatey*** to manage software installations. You can start the Python installation by running the following command in Command Prompt:

```
$ python --version  # Check installed Python version
```

For ***macOS***, you can use ***Homebrew*** to install Python and other tools. Check the installed version of Python with the command:

```
$ python3 --version  # Check installed Python 3
version
```

On ***Linux***, you can use package managers like ***apt*** or ***yum***, depending on your Linux distribution. Check the Python version with the following command:

```
$ python3 --version  # Check installed Python 3
version
```

Each operating system has its own methods and tools for installation and configuration, but ultimately, the goal is to ensure that you have a development environment ready for the ***platform_bot*** project. Choose the operating system that best fits your needs and comfort level with the available tools and software.

2.2.2 Ensuring System Requirements

Before starting the software installation for the ***platform_bot*** project, it is important to ensure that your system meets the minimum requirements to run the development environment and bot application smoothly. These requirements fall into two main categories: hardware and software.

Hardware Requirements will vary depending on the complexity of the project and the number of applications running simultaneously. For bot development using Django and Ngrok, the hardware requirements are generally not too high. However, here are some general recommendations to ensure optimal performance:

- ***Processor***: Most development tasks do not require a very powerful processor, but a modern multi-core processor will provide better performance, especially when running multiple applications or processes simultaneously.
- ***RAM***: At least 4 GB of RAM is recommended, though 8 GB or more will offer a better experience and allow you to run multiple applications without issues.
- ***Storage***: Development projects like this don't require much storage space, but ensure you have at least 10 GB of free space for your OS, software, and project files. An SSD (Solid State Drive) will improve data access speed and application load times.

- ***Internet Connection***: A stable internet connection is necessary to download software, perform updates, and test webhooks using Ngrok.

Software Requirements include the various tools and programs that need to be installed to begin development. The key software you will need includes:

- ***Python***: The latest version of Python 3.x should be installed on your system. Python is the programming language used to develop Django applications. You can check the installed version of Python with the following command:

```
$ python3 --version  # Check installed Python
version
```

- ***Package Manager (pip)***: Pip is Python's package manager, allowing you to install the necessary libraries and dependencies. Pip is usually included with Python installations. Check the installed version of pip with the command:

```
$ pip --version  # Check installed pip version
```

- ***Django***: The framework used to build your web application. Install Django using pip with the following command:

```
$ pip install django  # Install Django
```

- ***Ngrok***: A tool for creating an HTTP tunnel to your local server, allowing the webhook to function properly. You can download Ngrok from its official website and follow the installation instructions for your operating system. After installation, you can check the installed version of Ngrok with the following command:

```
$ ngrok version  # Check installed Ngrok
version
```

- ***Bot Framework***: Depending on the bot platform you choose (e.g., `python-telegram-bot` for Telegram), you will need additional libraries that can be installed using pip. For example, for `python-telegram-bot`, the command is:

```
$ pip install python-telegram-bot  # Install
python-telegram-bot
```

By ensuring that your hardware and software meet these requirements, you will be able to run the ***platform_bot*** smoothly and without significant technical issues. This step is crucial before proceeding with further installation and development of the project.

2.3 Installing Python And Virtual Environment

2.3.1 Installing Python

Python is the programming language that will serve as the foundation for developing the platform_bot project. Ensuring that Python is correctly installed on your system is a crucial first step. Here is a guide to downloading and installing Python on commonly used operating systems.

How to Download and Install Python

To begin, you need to download the latest version of Python from the official Python website. Visit the python.org site and select the Python 3 version that corresponds to your operating system. On the download page, you will find the stable and recommended version of Python.

- ***Windows***: Download the Windows installer and run the downloaded file. In the installation window, make sure to check the "Add Python to PATH" option before proceeding. This will make it easier to run Python from the Command Prompt. Click "Install Now" and follow the instructions to complete the installation.
- ***macOS***: Download the macOS installer from the Python site and open the .pkg file that was downloaded. Follow the installation prompts. Python will be installed in the default directory and can be accessed via the Terminal.
- ***Linux***: Most Linux distributions already include Python by default. However, if you need to install or update

Python, you can use the package manager appropriate for your distribution. For example, on Ubuntu, you can run:

```
$ sudo apt update  # Update the package list
$ sudo apt install python3  # Install Python 3
```

Checking the Installed Python Version

After the Python installation is complete, it's important to verify that Python has been installed correctly and check its version. This ensures that the installed version meets the requirements of your project.

- ***Windows***: Open the Command Prompt and run the following command:

```
$ python --version  # Check the installed Python
version
```

- ***macOS and Linux***: Open the Terminal and run the following command:

```
$ python3 --version  # Check the installed Python 3
version
```

This command will display the version of Python installed on your system. Make sure that the installed version is Python 3.6 or newer, as required for the platform_bot project.

With Python correctly installed and the appropriate version confirmed, you are ready to move on to the next step, which involves setting up a virtual environment to help manage project dependencies and keep your development environment clean.

2.3.2 Setting Up a Virtual Environment (Optional but Recommended)

To ensure a clean management of the platform_bot project, separate from other projects, we will use a virtual environment. A virtual environment is a tool that allows us to create an isolated environment for Python projects. This makes it easier to manage project-specific dependencies without interfering with the global Python settings on your system.

Explanation of Virtual Environment

A virtual environment is a directory that contains a standalone copy of the Python interpreter, as well as a folder for installing the packages required for the project. By using a virtual environment, you can:

- Avoid conflicts between the packages required by different projects.
- Keep the global Python system clean from packages that may only be needed for specific projects.
- Simplify dependency management by isolating the project environment.

How to Create and Activate a Virtual Environment

To create and activate a virtual environment, follow these steps:

1. ***Creating a Virtual Environment***
   Ensure you are in your project directory before creating

the virtual environment. Use the `venv` command provided by Python to create a virtual environment. Here is the command to create a virtual environment named `venv`:

```
$ python3 -m venv venv  # Create a virtual
environment named 'venv'
```

This command will create a new directory named `venv` within your project directory. This directory will contain a copy of the Python interpreter and subdirectories for the installed packages.

2. ***Activating the Virtual Environment***
   Once the virtual environment is created, you need to activate it to start working in that environment. The activation method varies depending on the operating system you are using.
   - ***Windows***: Run the following command in the Command Prompt to activate the virtual environment:

```
$ venv\Scripts\activate  # Activate the
virtual environment on Windows
```

   - ***macOS and Linux***: Run the following command in the Terminal to activate the virtual environment:

```
$ source venv/bin/activate  # Activate the
virtual environment on macOS and Linux
```

After the virtual environment is activated, your terminal prompt will change to indicate that you are in the virtual environment. For example, you will see the name of the virtual environment (e.g., `(venv)`) at the beginning of the command line.

Deactivating the Virtual Environment

To exit the virtual environment, simply run the following command:

```
$ deactivate  # Deactivate the virtual environment
```

Deactivating the virtual environment will return your terminal prompt to your global Python settings.

With the virtual environment set up and activated, you are ready to install project dependencies such as Django and other libraries without affecting the global Python settings on your system. This step lays the foundation for developing and managing the platform_bot project in a structured and separate manner.

2.4 Installing Django

After setting up Python and the virtual environment, the next step is to install Django and start your first Django project. Django is a powerful and flexible web framework, which will be the core of your platform_bot application development. Below, we will explain the process of installing Django and how to create your first Django project.

2.4.1 Using pip to Install Django

Pip is the package manager for Python used to install and manage third-party packages, including Django.

To install Django, ensure that your virtual environment is active. If the virtual environment is not yet activated, activate it first using the appropriate command for your operating system.

Here are the steps to install Django using pip:

Activate the Virtual Environment:
Make sure your virtual environment is active. If it is not active, activate it according to the previous instructions.

- ***On Windows:***

```
$ myenv\Scripts\activate
```

- ***On macOS and Linux:***

```
$ source myenv/bin/activate
```

Install Django:
Once the virtual environment is active, run the following command to install Django:

```
$ pip install django  # Install Django using pip
```

This command will download and install Django along with its dependencies into your virtual environment. This process requires an internet connection to download the necessary packages from PyPI (Python Package Index).

Verifying the Django Installation
After the installation is complete, it is important to verify that Django has been installed correctly and is ready to use. To verify the installation, you can check the version of Django installed with the following command:

```
$ django-admin --version  # Check the installed
version of Django
```

This command will display the version of Django currently installed in your virtual environment. Ensure that the installed version is the latest or matches the requirements for your platform_bot project.

With Django installed, you are now ready to start your first Django project. Next, you will learn how to create and configure a Django project for your platform_bot, and begin developing the features needed for your application. Make sure to follow the steps carefully to ensure your development environment is properly set up.

2.4.2 Creating Your First Django Project

After successfully installing Django, the next step is to create your first Django project. The Django project will serve as the main framework for your platform_bot application. This process includes creating the project and understanding the basic structure of the resulting project.

To create a new Django project, ensure that your virtual environment is active. Then, use the `django-admin` command to start creating the project. For example, to create a project named `platform_bot`, run the following command:

```
$ django-admin startproject platform_bot  # Create a
new Django project named 'platform_bot'
```

This command will create a new directory named `platform_bot`, which contains the basic structure of a Django project.

Basic Structure of a Django Project

After running the above command, Django will generate the basic directory structure for your project. This structure provides the necessary framework to develop your web application. Below is an overview of the basic Django project directory structure:

```
platform_bot/            # Main project directory
├── manage.py            # Script for managing the
Django project (e.g., running the server, making
migrations)
└── platform_bot/        # Django project directory
    ├── __init__.py      # Marks this directory as a
Python package
    ├── asgi.py          # Configuration for ASGI
(Asynchronous Server Gateway Interface)
```

```
    ├── settings.py      # Contains the project’s
settings and configurations
    ├── urls.py          # Handles URL routing for the
project
    └── wsgi.py          # Configuration for WSGI (Web
Server Gateway Interface)
```

- ***manage.py:*** This script is used for various administrative tasks such as running the development server, creating migrations, and starting Django apps. You will frequently use this script throughout your project development.
- ***platform_bot/:*** This is the project directory, which shares the same name as the project you created. Inside this directory are the key configuration files for the Django project:
    - ***`__init__.py:`*** Marks this directory as a Python package, allowing Django to recognize it as part of the project.
    - ***asgi.py:*** Contains configuration for ASGI, used for handling asynchronous applications and real-time web communication.
    - ***settings.py:*** This file holds the main settings and configurations for your Django project, such as database connections, static files, and app settings.
    - ***urls.py:*** Manages the URL routing for your project. Here, you will define URL patterns used to direct requests to the appropriate views.
    - ***wsgi.py:*** Contains configuration for WSGI, which is used for communication between the Django application and the web server.

With the basic project structure in place, you now have a foundation to start developing the platform_bot application. The next step is to configure and prepare your Django application for the specific needs of the project.

2.5 Ngrok Installation And Configuration

Ngrok is a tool that allows you to expose your local server to the internet via a secure public URL. This is particularly useful for testing web applications being developed in a local environment, especially when you need to showcase your project to others or test integrations with external services that require public access.

Ngrok is highly useful for accessing your local web application from outside your local network. It allows you to create HTTPS tunnels to your local server, enabling you to test and interact with your application in a broader environment, such as for webhook integration with your bot. Below are the steps to download and install Ngrok.

2.5.1 Downloading and Installing Ngrok

To download Ngrok, visit the official Ngrok website at ngrok.com/download. On this page, you will find download options for different operating systems, including Windows, macOS, and Linux.

- ***Windows:*** Choose the Windows version and download the ZIP file. Extract the downloaded ZIP file to a directory of your choice.
- ***macOS:*** Choose the macOS version and download either the ZIP or DMG file. If you download the ZIP file, extract it. If you choose the DMG file, open it and drag the Ngrok file into the Applications folder.
- ***Linux:*** Choose the Linux version and download the TAR file. Extract the TAR file using the following command:

```
$ tar -xvf ngrok-stable-linux-amd64.tar.gz  # Extract
the Ngrok TAR file
```

With Ngrok installed, you can now remotely access and test your platform_bot application. This is an essential step in verifying the functionality of your bot and ensuring that integration with external services is working properly.

2.5.2 Configuring and Running Ngrok

After installing Ngrok, the next step is to configure and run it to expose your local server to the internet.

Creating an Ngrok Account (Optional but Recommended)

Ngrok offers additional features like a dashboard and custom URLs if you sign up for a free account at https://ngrok.com/. After signing up, you will receive an authentication token that you can use to configure your Ngrok.

Adding the Authentication Token (Optional)

If you have an Ngrok account, you can add your authentication token to Ngrok by running the following command:

```
$ ngrok authtoken YOUR_AUTH_TOKEN
```

Replace `YOUR_AUTH_TOKEN` with the authentication token provided in your Ngrok account dashboard.

Running Ngrok

To expose your local Django server, which typically runs on `http://127.0.0.1:8000`, execute the following command:

```
$ ngrok http 8000
```

This command will open an HTTP tunnel on port 8000 and provide you with a public URL that can be accessed by anyone on the internet.

Ngrok Output

After running the command, you will see an output like this in your terminal:

```
Session Status                online
Account                       Your Name (Plan: Free)
Version                       3.0.0
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding
http://abcdef1234.ngrok.io -> http://localhost:8000
Forwarding
https://abcdef1234.ngrok.io -> http://localhost:8000
```

```
Connections                   ttl     opn     rt1
rt5     p50     p90
0.00    0.00    0.00    0.00    0.00
```

In this output, you will see two URLs, one using HTTP and another using HTTPS, for example:

```
Forwarding
http://abcdef1234.ngrok.io -> http://localhost:8000
Forwarding
https://abcdef1234.ngrok.io -> http://localhost:8000
```

You can use either of these URLs to access your local server from the internet.

Stopping Ngrok

To stop Ngrok, simply press `Ctrl + C` in the terminal.

By following these steps, you have successfully installed, configured, and run Ngrok to expose your local server to the internet. In the following chapters, we will begin creating your Django project and use Ngrok to test the application online.

2.5.3 Setting Up Ngrok for Webhooks

After successfully installing Ngrok, the next step is to configure Ngrok so it can be used for webhooks. Webhooks allow your application, such as a Telegram or WhatsApp bot, to receive data from external services in real-time. Ngrok will create an HTTPS tunnel to your local server, enabling the webhook to access your Django application from outside the local network.

Setting Up and Running Ngrok

First, ensure that your Django development server is running. Typically, you'll run the Django server on port 8000. To start the Django server, use the following command from your project directory:

```
$ python manage.py runserver 8000  # Run the Django development server on port 8000
```

Once the Django server is up and running, open a new terminal and run Ngrok to create a tunnel to port 8000. Use the following command to start Ngrok:

```
$ ./ngrok http 8000  # Create an HTTPS tunnel to port 8000
```

This command will generate an output displaying a public URL that can be used to access your local server. The URL will look something like this:

```
Forwarding                    https://<random-string>.ngrok.io -> http://localhost:8000
```

Connecting Ngrok with the Django Application

Now, you need to configure your Django application to receive webhooks from external services. For this example, let’s use a Telegram bot. You’ll need to configure the Telegram bot to send data to the Ngrok URL you obtained.

1. ***Obtaining the Telegram API Token***
   First, ensure that you have an API token for your Telegram bot. This token is obtained from the BotFather on Telegram.
2. ***Configuring the Webhook in Telegram***
   Use the API token and Ngrok URL to configure the webhook. Run the following command to set up the webhook using `curl` or a similar tool:

```
$ curl -F
"url=https://<random-string>.ngrok.io/webhook/"
https://api.telegram.org/bot<YOUR_BOT_TOKEN>/setWebho
ok
```

   Replace `<random-string>` with the random string from the Ngrok URL, and `<YOUR_BOT_TOKEN>` with your Telegram bot's API token.
3. ***Setting Up the Webhook URL in Django***
   In your Django application, create a view that will handle the data from the webhook. For example, create a `views.py` file in your bot app with the following code:

```
# bot/views.py
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import json

@csrf_exempt
def webhook(request):
    if request.method == 'POST':
        data = json.loads(request.body)
        # Process the webhook data from Telegram
        return JsonResponse({'status': 'ok'})
    return JsonResponse({'status': 'failed'})
```

Then, add the URL for this webhook view to your app's `urls.py` file:

```
# bot/urls.py
from django.urls import path
from .views import webhook

urlpatterns = [
    path('webhook/', webhook, name='webhook'),
]
```

With this setup, Ngrok will forward HTTPS requests to your local Django server, and Django will handle the webhook data from the Telegram bot.

Ngrok is now successfully configured to receive and forward webhooks to your Django application. You can monitor Ngrok's logs to check if the webhooks are being received correctly and inspect the responses provided by your Django application.

2.6 Installation And Configuration Of Bot Frameworks

In bot development, choosing the right framework is a crucial first step. The bot framework you select will determine how your bot interacts with users and other services.

2.6.1 Choosing a Bot Framework

There are several popular bot framework options available, each offering different features and integrations. Here are some commonly used bot frameworks:

1.Telegram Bot

Telegram is one of the platforms that highly supports bot development. It provides a powerful API and clear documentation to facilitate bot development. Telegram bots allow for complex interactions with users and can handle various types of content, such as text messages, images, and files.

Advantages:

- Comprehensive and flexible API
- Support for various types of content
- Active documentation and community

Getting Started: To get started with a Telegram bot, you need to register the bot through BotFather on Telegram and obtain an API token. You can then use this token to access the Telegram API from your Django application.

2.WhatsApp Bot

WhatsApp is a highly popular communication platform with millions of active users worldwide. To create a bot on WhatsApp, you can use services like Twilio or the WhatsApp Business API, which enable bot integration with WhatsApp.

Advantages:

- Access to a large user base
- Easy integration with Twilio and similar services
- Support for multimedia messaging

Getting Started: To use the WhatsApp API, you need to register through an API provider like Twilio and follow their guides to configure the bot and obtain API credentials.

3.Facebook Messenger Bot

Facebook Messenger is another platform that offers bot features. Messenger bots can be integrated with Facebook Pages and allow direct interaction with users through messages.

Advantages:

- Integration with Facebook Pages
- Support for text, images, and videos
- Advanced features for user interaction

Getting Started: To create a Messenger bot, you need to register as a Facebook Developer and configure the bot through the Facebook Messenger Platform. You will obtain an access token to interact with the Messenger API.

4.Other Platforms

In addition to the above bot frameworks, there are other platforms that may suit your needs, such as Slack, Microsoft

Teams, or Discord. Each platform has unique features and different APIs, so you can choose according to your goals and target audience.

Pros and Cons:

- ***Slack***: Integrated with collaboration tools, ideal for business applications.
- ***Microsoft Teams***: Integrated with Office 365, suitable for corporate environments.
- ***Discord***: Popular in the gaming community, supports many types of interaction.

Getting Started: For other platforms, follow their respective guides to register and configure bots according to the documentation provided by the service provider.

By choosing the appropriate bot framework, you can start developing an effective bot that suits your needs. After selecting a framework, the next step is to install and configure the SDK or API needed to interact with the platform of your choice.

2.6.2 Installing and Configuring the Bot Framework

After selecting the bot framework that fits your project's needs, the next step is to install the necessary libraries and configure the bot so it can run properly. In this section, we will focus on installing the libraries and setting up the initial configuration to begin bot development.

Installing Required Libraries

To start, you must ensure that all the libraries required by the bot framework are installed in your Django project environment. For example, if you choose Telegram as your bot platform, you can install a library that supports interaction with the Telegram API, such as `python-telegram-bot`.

First, make sure your virtual environment is active. If it is not, you can activate it with the following command:

```
$ source venv/bin/activate
```

Once the virtual environment is active, the next step is to install the necessary libraries. For a Telegram bot, install the `python-telegram-bot` library using `pip`:

```
$ pip install python-telegram-bot
```

Make sure the library is installed correctly by checking if it's listed with the following command:

```
$ pip freeze
```

If you are using another platform, such as WhatsApp via Twilio, you will need to install the `twilio` library:

```
$ pip install twilio
```

These installed libraries will allow you to interact with the API of the chosen bot platform, whether it’s Telegram, WhatsApp, or others.

2.7 Installation And Use Of Code Editors

A code editor is a crucial tool in software development, including for Django projects like the platform_bot. Choosing the right code editor can enhance productivity and make development more efficient. In this section, we will discuss installing and using a code editor, as well as features that support Django application development.

In software development, selecting the right code editor is essential to boost productivity. One highly flexible and powerful text editor is Vim. Although Vim is known for having a steep learning curve, once mastered, it can become an extremely efficient tool for development.

Visual Studio Code is also one of the most popular and widely used code editors in web development.

2.7.1 Choosing a Code Editor

There are many code editors you can choose for Python and Django development. Some popular ones include:

- ***Visual Studio Code (VS Code)***: A lightweight yet powerful code editor that supports many programming languages and has many useful extensions.
- ***PyCharm***: An IDE specifically designed for Python and Django, offering advanced features for web development.
- ***Sublime Text***: A fast and lightweight text editor with support for various plugins.
- ***Vim***: A terminal-based text editor that is very popular among software developers.

2.7.2 Vim

Vim is a terminal-based text editor that is very popular among software developers, especially those who often work in Unix/Linux environments. Vim supports many advanced features like syntax highlighting, split windows, macros, and more.

Installing Vim

To start using Vim, you need to install it first. Here are the installation steps for Vim on various operating systems:

1. ***Installation on Ubuntu/Debian***: If you are using a Debian or Ubuntu-based distribution, you can install Vim with the following command:

```
$ sudo apt update
$ sudo apt install vim
```

2. ***Installation on Fedora/CentOS***: For Red Hat-based distributions like Fedora or CentOS, use the following command:

```
$ sudo dnf install vim
```

3. ***Installation on macOS***: On macOS, you can install Vim using Homebrew:

```
$ brew install vim
```

4. ***Installation on Windows***: For Windows users, you can install Vim by downloading the installer from the official Vim website. Choose the installer that matches your Windows version and follow the installation instructions.
5. ***Verifying Installation***: After installation is complete, you can verify that Vim has been installed correctly by running the following command in the terminal:

```
$ vim --version
```

   The output will show the installed Vim version along with other configuration information.

Basic Vim Configuration

Once Vim is installed, you may want to configure it according to your development preferences. Here are some basic configurations you can do in Vim:

1. ***Creating a Configuration File***: Vim stores user configurations in a `.vimrc` file located in your home directory. If this file does not exist, you can create it with the following command:

```
$ touch ~/.vimrc
```

2. ***Basic Settings in `.vimrc`***: Here are some commonly used basic settings:

```
set number                      " Show line numbers
set tabstop=4                   " Set tab to 4 spaces
set shiftwidth=4                " Set indentation to 4
spaces
set expandtab                   " Convert tab to spaces
syntax on                       " Enable syntax
highlighting
set autoindent                  " Enable auto-
indentation
```

   To edit the `.vimrc` file, open it with Vim:

```
$ vim ~/.vimrc
```

   Then, insert the configuration above and save the file.

3. ***Basic Navigation in Vim***: Vim has different modes for navigating and editing text:
    - ***Normal Mode***: The default mode after opening Vim. You can navigate and perform text-editing operations.

- ***Insert Mode***: The mode for entering text. Enter this mode by pressing `i`, and return to Normal Mode by pressing `Esc`.
- ***Visual Mode***: The mode for selecting text. Enter this mode by pressing `v`.

4. ***Vim Usage Examples***:
   - ***Writing Text***: To start writing text, open a file with Vim:

     ```
     $ vim filename.txt
     ```

     Press `i` to enter Insert Mode, then type the text you want. Once done, press `Esc` to exit Insert Mode.
   - ***Saving and Exiting***: To save the file and exit Vim, press `Esc` to ensure you are in Normal Mode, then type `:wq` and press Enter.

     ```
     :wq
     ```

   - ***Selecting Text***: To select text, use Visual Mode. Press `v` to enter Visual Mode, use the arrow keys to select the desired text, then perform operations like copying (`y`), cutting (`d`), or pasting (`p`).
   - ***Undoing Changes***: To undo the last change, press `u` in Normal Mode. To redo the undone change, press `Ctrl + r`.

By following the guide above, you have installed and configured Vim with basic settings, and learned some frequently used commands. In the next stage, we will proceed to configure and use other editors that may be needed for your Django project development.

2.7.3 Installing Visual Studio Code

Visual Studio Code (VSCode) is one of the most popular and feature-rich open-source code editors, developed by Microsoft. VSCode supports various programming languages and has a powerful ecosystem of extensions, making it an ideal choice for developing Django projects.

1. ***Download Visual Studio Code***: Visit the official Visual Studio Code website at https://code.visualstudio.com/ and download the latest version for your operating system.
2. ***Install Visual Studio Code***:
   - ***On Windows***:
     After downloading the `.exe` installation file, run it and follow the on-screen instructions to complete the installation.
   - ***On macOS***:
     After downloading the `.dmg` file, open it and drag the Visual Studio Code icon to the Applications folder.
   - ***On Linux***:
     You can install Visual Studio Code using the

package manager or download the `.deb` or `.rpm` files from their website.

```
# For Debian-based distributions
$ sudo dpkg -i code_x.x.x_amd64.deb

# For RPM-based distributions
$ sudo rpm -i code-x.x.x.rpm
```

3. ***Setting Up Visual Studio Code for Django Development***: Once the installation is complete, open Visual Studio Code. To support Django development, you'll need to install a few additional extensions:
    - ***Python Extension***: This extension provides support for Python in Visual Studio Code, including IntelliSense, linting, and debugging.
    - ***Django Extension***: This extension adds specific features for Django, such as snippets and template support.

    To install these extensions, open Visual Studio Code, go to the Extensions tab (the box icon in the left sidebar), and search for "Python" and "Django". Click the install button on the appropriate extensions.
4. ***Configuring Visual Studio Code for Django Projects***:
    - ***Open Your Django Project***:
      You can open your Django project folder in Visual Studio Code by selecting `File > Open Folder` and choosing your project folder.
    - ***Set Up the Environment***:
      Ensure your virtual environment is active. You

can select the correct Python interpreter in Visual Studio Code by pressing `Ctrl+Shift+P` (Windows/Linux) or `Cmd+Shift+P` (macOS), and searching for "Python: Select Interpreter". Choose the interpreter located in your virtual environment.

- ***Using the Integrated Terminal***:
  Visual Studio Code provides an integrated terminal, allowing you to run Django commands directly from the editor. You can open the terminal by pressing `Ctrl+` (Windows/Linux) or `Cmd+` (macOS).

With Visual Studio Code installed and configured, you’re ready to start writing and managing code for your `platform_bot` project. Visual Studio Code will provide an efficient and enjoyable development environment, complete with features that support Django development effectively.

By installing the right extensions, VSCode becomes a powerful tool for developing Django projects. Once configured, you will have a development environment that allows you to start working on your Django project with high efficiency.

2.8 Setting Up Your Browser And Terminal/Command Line

Before starting the development of a Django project, it is important to ensure that you have a browser and terminal or command line ready to use. These tools are essential components in the development and testing cycle of a Django project. Here's a guide to set up your browser and terminal.

2.8.1 Browser

The browser is an essential tool for viewing the results of the web application you are developing. All interactions with the running Django project will be done through the browser.

1. ***Choose a Browser:***
   - ***Google Chrome:*** A popular browser that has many powerful Developer Tools.
   - ***Mozilla Firefox:*** Known for its rich development tools and good support for testing.
   - ***Microsoft Edge:*** Based on Chromium, it has similar development features to Chrome.
   - ***Safari:*** The default browser on macOS, it also provides good development tools.
2. ***Enabling Developer Tools:***
   - Developer Tools are part of the browser that allow you to inspect page elements, view the console, monitor the network, and debug JavaScript.
   - How to enable Developer Tools:

- ***Google Chrome/Edge:*** Press Ctrl + Shift + I (Windows/Linux) or Cmd + Option + I (macOS).
- ***Mozilla Firefox:*** Press Ctrl + Shift + I (Windows/Linux) or Cmd + Option + I (macOS).
- ***Safari:*** You need to enable the Developer menu through settings: go to Safari > Preferences > Advanced, then check "Show Develop menu in menu bar."

3. ***Testing Responsiveness:***
    - All modern browsers have features to test your site's responsiveness. This is important to ensure your site works well on various devices.
    - Access the responsive mode in Developer Tools to simulate how the site will look on devices like mobile phones and tablets.

2.8.2 Terminal/Command Line

The terminal or command line is a tool used to run commands, such as starting the Django server, managing Python packages, and working with the file system.

1. ***Terminal on Windows:***
    - ***Command Prompt:*** A built-in tool in Windows that can be accessed by searching for "cmd" in the Start Menu.
    - ***PowerShell:*** A more advanced tool with additional features, also found in the Start Menu.

- ***Windows Terminal:*** A modern terminal that supports multiple tabs and profiles, including PowerShell and Command Prompt. You can install it from the Microsoft Store.
  To open Command Prompt:

```
$ cmd
```

  To open PowerShell:

```
$ powershell
```

  To open Windows Terminal:

```
$ wt
```

2. ***Terminal on macOS:***
   - ***Terminal:*** A built-in application on macOS that can be accessed through Spotlight or the Applications > Utilities > Terminal folder.
     To open Terminal:

```
$ terminal
```

3. ***Terminal on Linux:***
   - ***Terminal:*** Available in all Linux distributions, usually accessed from the application menu or by pressing Ctrl + Alt + T.
     To open Terminal:

```
$ gnome-terminal
```

4. ***Tips for Using the Terminal:***
    - ***Navigating the File System:*** Use commands like `cd`, `ls`, and `pwd` to change directories and view content.
    - ***Editing Files:*** On many systems, you can use text editors like `nano`, `vim`, or `code` to edit files directly from the terminal.
    - ***Running Django Commands:*** After creating the Django project, many interactions will occur through the terminal, such as starting the server:

```
$ python manage.py runserver
```

5. ***Updating and Installing Packages:***
    - ***Windows:***

```
$ python -m pip install --upgrade pip
```

    - ***macOS/Linux:***

```
$ python3 -m pip install --upgrade pip
```

With your browser and terminal ready to use, you will have the necessary tools to begin developing your Django project. Both of these tools will be integral parts of your workflow, allowing you to run, test, and develop applications efficiently.

2.9 Environment Verification

2.9.1 Testing Django Installation

After completing all installation steps, it is essential to verify that Django has been installed correctly and is functioning without issues. In this section, we will run Django's built-in development server and check if the Django application can be accessed through a browser.

1.Running the Django Development Server

Django provides a built-in development server that is very useful for testing applications during the development process. To run this server, you must first ensure that you are in the Django project directory where you previously created the project.

If your virtual environment is not active, activate it first with the following command:

```
$ source venv/bin/activate
```

Once the virtual environment is active, navigate to your Django project directory. For example, if your Django project is named `platform_bot`, navigate into that folder:

```
$ cd platform_bot
```

Ensure that the structure of your Django project is correct, with essential files and folders like `manage.py`, `settings.py`, and `urls.py` present. Then, run the Django development server using the following command:

```
$ python manage.py runserver
```

This command will start the development server at the address `http://127.0.0.1:8000/` by default. You will see output in the terminal confirming that the server is running successfully, as follows:

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

System check identified no issues (0 silenced).
September 16, 2024 - 14:30:10
Django version 4.2.0, using settings
'platform_bot.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

2.Checking Access to the Application

After the development server is running, open your browser and access the address `http://127.0.0.1:8000/`. If the Django installation is successful and the server is running well, you will see the Django welcome page displaying the message "The install worked successfully! Congratulations!".

This page indicates that Django has been installed and configured correctly. If you do not see this page or if there are error messages in the terminal or browser, double-check your configuration and ensure all dependencies are installed correctly.

If the server cannot be accessed or if errors appear, here are some troubleshooting steps:

- Ensure that port 8000 is not being used by another application.
- Make sure all libraries and dependencies have been installed correctly in the virtual environment.
- Check the error logs in the terminal for more specific messages.

With the successful testing of this server, the Django development environment is ready to be used for your `platform_bot` project. You can now start building applications and developing various desired features.

2.9.2 Testing Ngrok and Webhook

After successfully testing Django, the next step is to ensure that Ngrok is functioning properly and can be used to connect your local Django application to the internet via webhook. Ngrok is very helpful when developing bots that require webhooks, such as Telegram or WhatsApp bots, allowing external services to communicate with your local server.

Ensuring Ngrok Works Well
First, we will run Ngrok to ensure that the tunnel to the local Django server can be opened and accessed from the internet. Ngrok allows us to create a public URL that directs to the local development server on a specific port.

Run the following command to start Ngrok and create a tunnel to the Django development server running on port 8000:

```
$ ngrok http 8000
```

Ngrok will output information in the terminal displaying the public URL that can be accessed from the internet, for example:

```
Session Status                online
Account                       your_email@example.com
(Plan: Free)
Version                       2.3.40
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding
http://abcdef1234.ngrok.io -> http://localhost:8000
Forwarding
https://abcdef1234.ngrok.io -> http://localhost:8000
```

Here, the public URL `https://abcdef1234.ngrok.io` is the address that can be used to access the local Django server via the internet. Make sure to save this URL, as it will be used for the bot's webhook.

Testing Webhook for the Bot

Webhooks allow the bot to communicate with your server when a message or command is sent by a user. To test the webhook, we will connect the created Ngrok URL with your bot application.

If you are using a Telegram bot, you can set the bot's webhook using the Telegram API command. For example, use the following command to connect the bot's webhook to the Ngrok URL:

```
$ curl -F "url=https://abcdef1234.ngrok.io/webhook/"
https://api.telegram.org/bot<YOUR_BOT_TOKEN>/setWebho
ok
```

Replace `<YOUR_BOT_TOKEN>` with your Telegram bot token, and ensure that the `/webhook/` endpoint in your Django application is configured to handle messages from the bot. If the webhook is successfully set, Telegram will send data to the `/webhook/` endpoint every time there is a new message.

Next, open the Django terminal and ensure that the development server is running. If the webhook is functioning properly, you will see requests from Telegram coming into your server every time the bot receives a message. These requests can be viewed in the Django terminal or in the Ngrok interface at `http://127.0.0.1:4040`, which displays request logs.

An example of request logs in Ngrok might look like this:

```
POST /webhook/ 200 OK
```

If you see logs like the above, it means the webhook has been successfully set, and your bot can communicate with the Django server through Ngrok.

If there are errors, check the Ngrok and Django terminal logs to see if there are any mistakes in the webhook endpoint or bot configuration.

With the successful completion of this testing, you have verified that Ngrok and the webhook are working properly. Your bot is now ready to receive and respond to messages from platforms such as Telegram or WhatsApp.

Chapter Conclusion

In this chapter, we have gone through several important stages to set up the development environment and run the platform_bot project. From installing Python, setting up the virtual environment, to installing Django, and configuring tools like Ngrok and code editors, each step is designed to ensure that you have a solid foundation for continuing the development of webhook-based bots with Django.

Here’s a summary of the steps we’ve taken:

1. ***Installing Python***
   We began by installing Python, which serves as the foundation for developing Django applications and bots. The version of Python used must meet the requirements of Django and the chosen bot framework.
2. ***Setting Up the Virtual Environment***
   To maintain consistency and prevent conflicts between project dependencies, we set up a virtual environment. This ensures that each project has separate dependencies that do not interfere with one another.
3. ***Installing Django***
   Once the virtual environment is ready, we installed Django, the main framework we will use to build the bot platform. This installation was verified by running the development server and ensuring that the Django application could be accessed through a browser.

4. ***Setting Up Ngrok***
   Ngrok is used to make our local Django server accessible from the internet. With Ngrok, we can connect webhooks from platforms like Telegram or WhatsApp to our local server, allowing the bot to receive and respond to messages from users.
5. ***Testing Ngrok and Webhooks***
   After Ngrok is connected, we ensure that the webhooks function properly, and the bot can receive requests from external platforms. Ngrok helps connect the development server with bot services, and testing the webhooks confirms that this connection runs smoothly.

Tips and Suggestions for Troubleshooting:

1. ***Ngrok Connection Issues:***
   If Ngrok cannot create a tunnel or cannot be accessed from outside, ensure that your firewall or network configuration does not block access to the used port. Additionally, check if your local server is running on the correct port (8000 for Django by default).
2. ***Webhook Errors:***
   If the webhook is not functioning or the bot does not respond to messages, double-check the URL used when setting up the webhook, ensuring that the provided Ngrok URL is an HTTPS URL. Also, check the webhook endpoint in the Django application to ensure it is configured correctly to receive requests from the bot platform.

3. ***Incompatible Dependencies:***
   If you encounter issues installing Python packages or other dependencies, check the versions of Python and Django you are using. Compatibility issues often arise due to mismatched package versions. Try using a virtual environment to ensure each project has a clean and isolated environment.
4. ***Manual Testing of the Django Server:***
   If the Django development server cannot be run, check for issues within the code or project configuration. Verify the `settings.py` file to ensure all configurations are correct, and examine the terminal logs for error messages that may help in troubleshooting.

With all the steps completed, you now have a strong foundation to continue developing your bot platform. Be sure to conduct regular testing during the development process and follow best practices to maintain the quality and security of your project.

Chapter 3 - Understanding and Using Django Admin

In this chapter, we will explore Django Admin, a very useful tool for data management in Django applications. Django Admin serves as a built-in administration panel that allows developers and administrators to manage models and data easily and efficiently. We will begin with an introduction to what Django Admin is, including its benefits and key features. Next, we will discuss the steps to create a superuser that will provide full access to the admin panel.

After that, we will look at how to register models in Django Admin and how we can customize the appearance and add additional features to enhance the user experience. Additionally, we will discuss security aspects, including how to manage user permissions and restrict access to the admin panel. Finally, we will identify common issues that may arise when using Django Admin and solutions to address them.

With a deep understanding of Django Admin, we can maximize the potential of our Django applications in data management, making them more efficient and user-friendly.

3.1 What Is Django Admin?

Django Admin is a built-in administration panel provided by the Django framework. As an integral part of Django, Admin is designed to facilitate the management of data and models used in web applications. With an intuitive and user-friendly interface, Django Admin enables developers and administrators to perform various administrative tasks, such as adding, editing, or deleting data, without needing to write additional code.

The main benefit of using Django Admin is its ability to speed up the application management process. With just a few configuration steps, users can access and manage the models that have been defined in the application. For example, after creating a model, simply adding a few lines of code to the `admin.py` file will make that model immediately available in the Admin interface. The use of Django Admin reduces the need to create custom user interfaces for data management, which often consumes time and resources.

When using Django Admin, users gain access to advanced features such as search, filtering, and user access management. These features are particularly useful for organizing large amounts of data and ensuring that only certain users have the rights to perform specific actions. Furthermore, Django Admin also allows users to customize the appearance and functionality according to their needs, providing greater flexibility in managing applications.

With a thorough understanding of Django Admin, users can maximize its potential and enhance efficiency in managing the web applications they develop. Throughout this chapter, we will explore how to effectively set up and use Django Admin, from creating a superuser to personalizing the interface.

3.2 Key Features Of Django Admin

Django Admin is equipped with various advanced features that support efficient data management. These features not only simplify the administrator's task of managing models but also facilitate user interaction with data within the application. Here are some of the main features provided by Django Admin.

One of the most important features of Django Admin is model management. Once a model is defined in the application, users can easily manage it through the Admin interface. By adding a few lines of code in the `admin.py` file, the model will automatically be available in Django Admin. For example, to register the Blog model that has been created, the user only needs to write the following code:

```
# admin.py
from django.contrib import admin
from .models import Blog  # Importing the Blog model

admin.site.register(Blog)  # Registering the Blog
model to Django Admin
```

After this code is added, users can immediately access the Blog model through the Admin panel, performing operations such as adding, editing, or deleting entries with ease.

Django Admin also offers a very useful search feature. With this feature, users can quickly and efficiently search for specific entries in the model. Developers can add search functionality to the model by defining the `search_fields` attribute within the Admin class. For instance, if the Blog model has `title` and `content` attributes, the user can add the following code to enable searching by both attributes:

```
# admin.py
class BlogAdmin(admin.ModelAdmin):
    search_fields = ['title', 'content']  # Defining
the fields that can be searched

admin.site.register(Blog, BlogAdmin)  # Registering
the Blog model with search configuration
```

The filter feature is also one of the main strengths of Django Admin. With this feature, users can filter entries based on specific criteria, making it easier to search for relevant data among large amounts of information. Developers can add filtering features using the `list_filter` attribute in the Admin class. For example, if the Blog model has a `published_date` field, the user can set up a filter based on the publication date by adding the following code:

```
# admin.py
class BlogAdmin(admin.ModelAdmin):
    list_filter = ['published_date']  # Defining the
fields that can be used for filtering
```

```
admin.site.register(Blog, BlogAdmin)  # Registering
the Blog model with filter configuration
```

These features make Django Admin a very powerful and flexible tool for data management. Additionally, Django Admin also allows users to manage access rights more effectively. With user and group management, administrators can determine who has access to specific parts of the application, thereby enhancing security and data management.

Overall, the features offered by Django Admin are extremely helpful in accelerating the web application management process. By understanding and utilizing these features, users can be more effective in managing data, improving productivity, and ensuring the security of the applications they develop. In the following chapters, we will explore in greater depth how to practically implement these features in Django applications.

3.3 Creating A Superuser

In this section, we will discuss the steps to create a superuser, which is an important step for accessing Django Admin with full authorization. A superuser is a user account with the highest access rights in the system, allowing the user to perform all actions in Django Admin, including model management, permission settings, and other data management tasks.

3.3.1 What Is a Superuser?

A superuser in Django functions as the main administrator who has full control over all aspects of the application. In other words, a superuser can access and manage all models registered in Django Admin, as well as perform actions that may be restricted for regular users. This is particularly useful in the context of applications with many users or complex data, where only certain individuals should have access to make significant changes.

In the context of Django Admin, a superuser is a user with the highest access rights, capable of managing all aspects of the application. The role of a superuser is crucial as it can perform administrative actions without limits, including managing models, users, and other application configurations. Superusers are typically used by developers or system administrators to oversee and manage data, ensuring that the application runs smoothly.

When creating a superuser, the user will be granted full access to the Django Admin interface. This means they can add, edit, and delete entries in all registered models. For example, if the application has a Blog model, the superuser can create and manage posts, edit categories, and view all data related to that model. Thus, the superuser acts as the primary manager of the application, ensuring that all administrative processes can be carried out efficiently.

The superuser's role also includes managing other users. With the access rights they possess, superusers can create new users, modify their access permissions, and add them to specific groups. This is particularly important in the context of larger applications, where there may be many users with varying levels of access. Therefore, the superuser acts as a supervisor, ensuring that each user has the appropriate access rights and is accountable for data management.

Overall, the existence of a superuser in Django Admin is key to effective application management. With a clear understanding of its role and function, users can fully leverage the capabilities of Django Admin to keep data secure and well-organized. In the next sub-section, we will discuss the concrete steps to create a superuser and how to log in to the Django Admin interface.

3.3.2 Steps to Create a Superuser

To fully leverage the features offered by Django Admin, the first step is to create a superuser. This process is very simple and can be completed in just a few steps. Here is a step-by-step guide to creating a superuser using the terminal.

First, ensure that the Django server is running and that the database has been configured correctly. If you are using SQLite, the database is usually created automatically when the project is first run. However, if you are using another database such as PostgreSQL or MySQL, make sure that the connection is properly set up in the `settings.py` file.

After confirming that all requirements have been met, open the terminal and navigate to your Django project directory. In the project directory, run the following command to start the superuser creation process:

```
$ python manage.py createsuperuser  # Command to
create a superuser
```

After running this command, the system will prompt you for some important information to create the superuser account. First, you will be asked to enter a username. This username will be used to log in to Django Admin, so choose a name that is easy to remember and unique.

Next, the system will ask for an email address. Although this email is not always required for access to Django Admin, it is highly recommended to provide a valid email. This email can be used for password recovery in the future or to receive notifications related to the account.

After entering the email, you will be prompted to create a password. This password should be strong enough to maintain the security of the application. Django will validate the strength of the password you create, so be sure to follow the security guidelines provided. If the password you enter does not meet the criteria, you will be asked to enter a new password.

Here is an example of the terminal interaction when creating a superuser:

```
Username (leave blank to use 'admin'): admin  #
Entering username
Email address: admin@example.com  # Entering email
Password:  # Entering password
Password (again):  # Re-entering password for
confirmation
```

After all the information has been entered and the password has been verified, the superuser will be successfully created. You will see a confirmation message stating that the superuser has been created successfully.

With the superuser created, you can now log in to Django Admin using the username and password you specified. Open your browser and enter the following URL:

```
http://localhost:8000/admin  # URL to access Django
Admin
```

Once the login page opens, enter the username and password you created earlier. If all the information is correct, you will be directed to the Django Admin dashboard, where you can start managing the models and data of the application.

Creating a superuser is a crucial first step to effectively utilize Django Admin. With the superuser ready, you now have full control over the application and can perform various administrative actions with ease. Next, we will discuss how to log in to Django Admin and explore its interface.

3.3.3 Logging into Django Admin with a Superuser

After successfully creating a superuser, the next step is to log in to Django Admin to start managing the application. This login process is very simple and allows you to access all the features available in the administration interface.

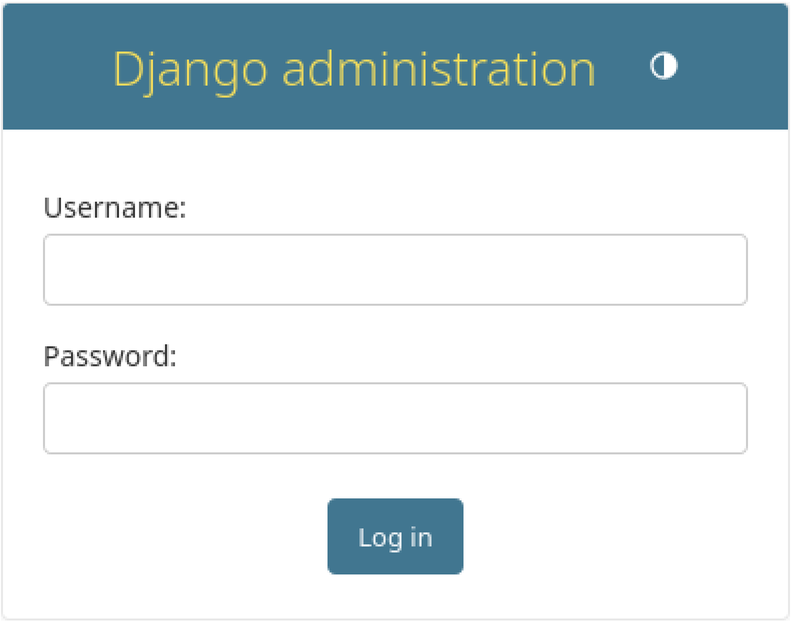
First, ensure that the Django server is running. If you haven't started it yet, open the terminal and navigate to your Django project directory. Then, run the following command to start the server:

```
$ python manage.py runserver  # Command to run the Django server
```

Once the server is active, open your browser and type the following URL to access the Django Admin login page:

```
http://localhost:8000/admin  # URL to access Django Admin
```

After the login page opens, you will see a form requesting your username and password. Enter the username and password you created during the superuser creation process. An example of the login form is as follows:



After entering the correct information, click the "Log in" button to enter Django Admin. If all the data entered is correct, you will be directed to the main dashboard of Django Admin.

The initial view of this dashboard provides an overview of the models that have been registered in the application. On the left side of the screen, there is a navigation menu listing all available models. You will see categories based on the models you have registered, such as Blog, Users, or other models. Each category can be expanded to display the entries that have been created within that model.

At the top, there are several navigation options, including the ability to log out and access Django documentation. This dashboard also provides search and filter features that allow you to quickly find specific data.

One interesting aspect of the Django Admin interface is its simplicity. Despite having many features, its design remains clean and intuitive, making it easy for new users to understand how to manage data effectively. On the right side, there are buttons that allow you to add new entries for each model, making it easier to perform basic operations such as adding, editing, or deleting data.

Once logged in, you can explore various sections of Django Admin, from managing data, performing searches, to using the filter features mentioned earlier. With an active superuser, you have full control over the application, opening up many possibilities for more efficient management.

With these steps, you have successfully logged into Django Admin using a superuser. Next, we will discuss how to add models to Django Admin for better management.

3.4 Adding Models To Django Admin

After successfully logging into Django Admin, the next step is to register the models you have created so that they can be managed through the administration interface. This process is quite simple and involves changes to the `admin.py` file within the relevant application.

3.4.1 Registering Models with Django Admin

Models in Django are representations of the data structures that will be used in the application. Once the models are defined, you need to inform Django that these models should be available in Django Admin. To do this, open the `admin.py` file located in the application folder. If this file does not exist, you can create it.

For example, if we have a model named `Blog` defined in the `models.py` file, we need to register this model in the `admin.py` file. Here is a simple example of the necessary code:

```
# admin.py
from django.contrib import admin
from .models import Blog  # Importing the Blog model

admin.site.register(Blog)  # Registering the Blog
model with Django Admin
```

In the code above, we first import the `admin` class from `django.contrib`. Then, we also import the `Blog` model that we previously created. By using the `admin.site.register()` method, we register the `Blog` model with Django Admin.

After saving the changes to the `admin.py` file, you need to ensure that the Django server is still running. If it is not, run the following command in the terminal:

```
$ python manage.py runserver  # Running the Django
server
```

Once the server is running, reopen the Django Admin interface in your browser with the URL:

```
http://localhost:8000/admin  # URL to access Django
Admin
```

After logging in, you will see the `Blog` model you just registered appear in the navigation menu on the left. Clicking on that model will take you to a page displaying all entries in the `Blog` model. If there are currently no entries, you will see an option to add a new entry.

With the model registered, you can now perform various operations such as adding, editing, and deleting blog posts directly from the Django Admin interface. This greatly simplifies data management and enhances efficiency in application development.

Registering models with Django Admin is an important step that allows administrators to manage data effectively. By understanding how to register models, you can easily extend the functionality of your application and leverage the power of Django Admin to improve productivity in data management. In the following subsection, we will discuss how to customize the appearance of models in Django Admin to better suit the needs of your application.

3.4.2 Customizing Model Appearance in Admin

After registering models in Django Admin, the next step is to customize the appearance of these models to make them more user-friendly and informative. Django provides several features that allow us to control the display and functionality of models in the Admin interface, such as `list_display`, `list_filter`, and `search_fields`. By using these features, we can enhance the user experience when managing data.

To get started, reopen the `admin.py` file where the `Blog` model has been registered. To customize the model's appearance, we need to create a subclass of `admin.ModelAdmin` and override the attributes we want to modify. Here is an example of how to add control over the appearance of the `Blog` model:

```
# admin.py
from django.contrib import admin
from .models import Blog  # Importing the Blog model

class BlogAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'created_at')
# Displaying specific columns
    list_filter = ('author', 'created_at')  # Adding
filters based on author and date
    search_fields = ('title', 'content')  # Enabling
search by title and content

admin.site.register(Blog, BlogAdmin)  # Registering
the Blog model with customization
```

In the code above, we define the `BlogAdmin` class, which is a subclass of `admin.ModelAdmin`. By using the `list_display` attribute, we can specify which columns we want to display in the list of `Blog` entries. In this example, we display the title, author, and creation date. This provides clearer and more comprehensive information to users when viewing the list of entries.

Next, the `list_filter` attribute allows us to add filters on the right side of the list page. With these filters, users can easily filter entries by author and date. This is particularly useful when there are many entries in the model.

Finally, the `search_fields` attribute provides better search capabilities by allowing users to search for entries by title and content. This is very helpful when looking for specific information among many entries, speeding up the data management process.

After making these changes, be sure to save the `admin.py` file and, if necessary, restart the Django server for the changes to take effect. You can do this by running the following command in the terminal:

```
$ python manage.py runserver  # Restarting the Django server
```

Once the server is active, reopen Django Admin in your browser and navigate to the `Blog` model. You will see the updated appearance according to the settings you have specified. With a customized view, data management becomes much more efficient and easier for administrators.

Utilizing features like `list_display`, `list_filter`, and `search_fields` is key to creating a user-friendly administration interface.

With this approach, we can provide ease to users in managing data and improve productivity within the application. In the next subsection, we will discuss how to further modify the appearance of Django Admin to better suit the needs of the application.

3.5 Customizing Django Admin

Django Admin is a very useful tool for managing data, but often we want to customize its interface to better fit the identity of our application or to enhance the user experience. In this subsection, we will discuss how to customize the appearance of Django Admin by adding simple CSS attributes and changing the admin templates.

3.5.1 Changing the Appearance of Django Admin

One of the easiest ways to customize the appearance of Django Admin is by adding custom CSS. To do this, we need to create a new directory in our application that will store the CSS files. First, create a folder named `static` in your application if it doesn't already exist. Inside this folder, create a `css` folder to store our custom CSS files.

For example, we can create a CSS file named `admin_custom.css`. Here are the steps you can follow:

Create Folder Structure: Inside the application directory, create the following folder structure:

```
your_app/
    static/
        css/
            admin_custom.css  # Custom CSS file
```

Add Custom Styles: Open the `admin_custom.css` file and add some custom styles. For example, we can change the background color and font:

```
/* admin_custom.css */
body {
    background-color: #f5f5f5;  /* Changing the
background color */
    font-family: 'Arial', sans-serif;  /* Changing
the font */
}

.dashboard-header {
    background-color: #007bff;  /* Changing the
header color */
    color: white;  /* Changing the header text color
*/
}
```

Including Custom CSS in Django Admin: Next, we need to tell Django to include this custom CSS file in the admin interface. We can do this by adding the following code to the `admin.py` file:

```
# admin.py
```

```
from django.contrib import admin
from django.utils.html import format_html
from django.conf import settings

class CustomAdminSite(admin.AdminSite):
    site_header = 'My Custom Admin'  # Changing the
admin page title

    def get_urls(self):
        urls = super().get_urls()
        custom_css_url =
f"{settings.STATIC_URL}css/admin_custom.css"  # URL
of the CSS file
        return [format_html('<link rel="stylesheet"
type="text/css" href="{}">', custom_css_url)] + urls

admin_site = CustomAdminSite(name='custom_admin')  #
Using CustomAdminSite
```

With the code above, we create a new class named `CustomAdminSite` that allows us to customize the admin page title and include the custom CSS file we created. We also use `get_urls` to include the CSS in the admin interface.

Registering Models with CustomAdminSite: Finally, we need to ensure that our models are registered using the `CustomAdminSite` we created. For example, if we have a `Blog` model, we can register it like this:

```
# admin.py
from .models import Blog  # Importing the Blog model

admin_site.register(Blog)  # Registering the Blog
model with CustomAdminSite
```

After saving these changes, run the Django server if it's not already running:

```
$ python manage.py runserver  # Starting the Django
server
```

Then, reopen the Django Admin interface in your browser and check if the changes have been applied. You will see a more customized view that matches your desired style.

Customizing the appearance of Django Admin with custom CSS is an effective way to create a more attractive interface that aligns with your application's branding. With these steps, you can add a personal touch to the admin panel, ultimately enhancing the user experience. In the next subsection, we will discuss how to add custom actions in Django Admin to improve its functionality.

3.5.2 Adding Custom Actions in Django Admin

One of the great features offered by Django Admin is the ability to add custom actions. These actions allow administrators to perform specific operations in bulk on one or more model entries. For example, we can create an action to delete multiple items at once or perform more complex custom actions according to the needs of the application.

To add a custom action to a model in Django Admin, we will continue with the `Blog` model that we registered earlier. Let's create a custom action that will delete all entries selected by the user. Here are the steps you need to follow:

Open the admin.py File: Make sure you are in the `admin.py` file where the `Blog` model is registered.

Define the Custom Action Function: We need to define a function that will run when the action is selected. This function will accept two parameters: `modeladmin` and `queryset`. Here is an example function to delete the selected entries:

```
# admin.py
from django.contrib import admin
from django.contrib import messages
from .models import Blog  # Importing the Blog model

def delete_selected_blogs(modeladmin, request,
queryset):
    # Deleting the selected entries
    deleted_count, _ = queryset.delete()  # Deleting
entries
    messages.success(request, f'{deleted_count}
entries successfully deleted.')  # Displaying success
message

delete_selected_blogs.short_description = 'Delete
selected entries'  # Description for the action
```

In the code above, we define the function `delete_selected_blogs` that will delete the selected entries. This function also uses `messages` to provide feedback to the user after the action is executed.

Add the Custom Action to ModelAdmin: Next, we need to add this custom action function to the `BlogAdmin` class we created earlier. Here are the updates you need to make:

```
# admin.py
class BlogAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'created_at')
# Displaying specific columns
```

```
    list_filter = ('author', 'created_at')  # Adding
filters
    search_fields = ('title', 'content')  # Enabling
search
    actions = [delete_selected_blogs]  # Adding
custom action to admin

admin.site.register(Blog, BlogAdmin)  # Registering
the Blog model with customization
```

By adding `actions = [delete_selected_blogs]`, we are telling Django Admin to include the custom action we just created.

Run the Server and Test the Action: After making these changes, save the `admin.py` file and run the Django server if it’s not already running:

```
$ python manage.py runserver  # Starting the Django
server
```

Now, open the Django Admin interface in your browser and navigate to the `Blog` model. Select several entries by checking the boxes next to each entry, then choose the action "Delete selected entries" from the actions dropdown above the list. After clicking "Go," all selected entries will be deleted, and you will see a success message at the top of the page.

Adding custom actions in Django Admin allows administrators to perform specific tasks efficiently and enhances control over the data. With actions like "Delete selected entries," users can manage data more easily and quickly, which is a significant advantage in application management.

3.6 Additional Features Of Django Admin

In this section, we will discuss additional features that can enhance the functionality of Django Admin, such as using inline forms to edit related models directly on the admin page.

3.6.1 Inline Forms in Django Admin

One very useful feature in Django Admin is the ability to use "Inline Forms." This feature allows users to display and edit related models directly from the admin page without needing to open a separate page. This is particularly useful when we have relationships between models, such as a `Blog` model that may have several related comments.

To demonstrate the use of inline forms, let's assume we have two models: `Blog` and `Comment`. The `Comment` model will have a one-to-many relationship with the `Blog` model, where one blog can have many comments. Here are the steps to implement this feature.

Define the Comment Model: First, we need to ensure that we have a `Comment` model related to the `Blog` model. Here's an example of how this model can be defined:

```
# models.py
from django.db import models
```

```
class Blog(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    created_at = 
models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

class Comment(models.Model):
    blog = models.ForeignKey(Blog, 
on_delete=models.CASCADE, related_name='comments')  # 
Relationship with Blog
    author = models.CharField(max_length=100)
    text = models.TextField()
    created_at = 
models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f'Comment by {self.author} on 
{self.blog}'
```

In this example, the `Comment` model has a `blog` field that is a foreign key pointing to the `Blog` model, making each comment related to one blog.

Add Inline Form in Admin: Next, we need to add an inline form for the `Comment` model in the admin panel of the `Blog` model. Open the `admin.py` file and add the following code:

```
# admin.py
from django.contrib import admin
from .models import Blog, Comment  # Importing Blog 
and Comment models

class CommentInline(admin.TabularInline):  # Using 
TabularInline for table view
    model = Comment  # Model to be used
    extra = 1  # Number of empty forms to display
```

```
class BlogAdmin(admin.ModelAdmin):
    list_display = ('title', 'created_at')  # 
Displaying specific columns
    inlines = [CommentInline]  # Adding inline form 
for comments

admin.site.register(Blog, BlogAdmin)  # Registering 
the Blog model with customization
```

In the code above, we create a `CommentInline` class that is a subclass of `admin.TabularInline`. This class will display the form for the `Comment` model within the admin page for the `Blog` model. We also add the `extra` parameter to specify the number of empty forms we want to display.

Run the Server and Test Inline Forms: After saving the changes in the `admin.py` file, run the Django server if it's not already running:

```
$ python manage.py runserver  # Starting the Django 
server
```

Next, open the Django Admin interface in your browser and navigate to the `Blog` model. When creating or editing a blog entry, you will now see a form to add comments below the main blog form. This allows users to add new comments directly without leaving the page.

The use of inline forms is very advantageous when we have models with complex relationships, as it makes data management more efficient and organized. In this way, users can easily view and edit related data in one view, enhancing the overall user experience.

3.6.2 Creating Custom Filters

Django Admin not only provides the ability to display and manage data but also allows us to create custom filters. These custom filters enable administrators to easily filter data based on specific criteria relevant to the application. This is especially useful when we have many entries and want to focus on a particular subset of data.

Let's see how to create custom filters for the `Blog` and `Comment` models. For example, we want to create a custom filter that allows users to filter comments based on the author.

Creating a Custom Filter Class: To create a custom filter, we need to define a class that inherits from `admin.SimpleListFilter`. Here's an example of how to define this custom filter:

```
# admin.py
from django.contrib import admin
from .models import Blog, Comment  # Importing Blog
and Comment models

class CommentAuthorFilter(admin.SimpleListFilter):
    title = 'Comment Author'  # Title of the filter
displayed in admin
    parameter_name = 'author'  # Name of the
parameter used in the URL

    def lookups(self, request, model_admin):
        authors = set([c.author for c in
Comment.objects.all()])  # Getting unique list of
comment authors
        return [(author, author) for author in
authors]  # Returning a list of tuples (value, label)

    def queryset(self, request, queryset):
        if self.value():  # If a filter value is
selected
            return
queryset.filter(author=self.value())  # Returning the
filtered queryset
        return queryset  # Returning the queryset
without filters
```

In the code above, the `CommentAuthorFilter` class defines a filter to filter comments based on the author. The `lookups` method generates a list of unique authors from all existing comments, while the `queryset` method returns the filtered queryset based on the selected author.

Adding the Custom Filter to Admin: Next, we need to add this custom filter to the `BlogAdmin` class. Here are the updates to make in the `admin.py` file:

```
# admin.py
class BlogAdmin(admin.ModelAdmin):
    list_display = ('title', 'created_at')  #
Displaying specific columns
    list_filter = (CommentAuthorFilter,)  # Adding
custom filter

admin.site.register(Blog, BlogAdmin)  # Registering
the Blog model with customization
```

By adding `list_filter = (CommentAuthorFilter,)`, we inform Django Admin to include the custom filter we just created in the admin panel of the `Blog` model.

Run the Server and Test the Custom Filter: After saving the changes in the `admin.py` file, run the Django server if it’s not already running:

```
$ python manage.py runserver  # Starting the Django
server
```

Now, open the Django Admin interface in your browser and navigate to the `Comment` model. On the right side, you will see a new filter named "Comment Author." When users select an author from the list and apply the filter, Django Admin will filter the comments to display only those written by that author.

By adding custom filters in Django Admin, we enhance the functionality and usability of the admin interface. Users can quickly find relevant data based on specific criteria, which is crucial when dealing with large amounts of data.

3.6.3 Exporting Data from Django Admin

One of the most useful features in Django Admin is the ability to export data directly from the admin page, such as into a CSV format. This allows administrators to retrieve data from the application and use it outside the system, such as for further analysis, reporting, or other data processing needs.

Let's look at how to add an export data feature to the `Comment` model we discussed earlier.

Creating a CSV Export Function: First, we need to create a function that will export data into CSV format. This function will take a queryset from the model and convert it into CSV format. Here's an example of how this function can be defined:

```
# admin.py
import csv
from django.http import HttpResponse
from django.contrib import admin
from .models import Comment  # Importing the Comment
model

class CommentAdmin(admin.ModelAdmin):
    list_display = ('author', 'text', 'created_at')
# Displaying specific columns

    def export_as_csv(self, request, queryset):  #
Function to export data
        response =
HttpResponse(content_type='text/csv')  # Setting the
content type to CSV
        response['Content-Disposition'] =
'attachment; filename="comments.csv"'  # Specifying
the filename

        writer = csv.writer(response)  # Creating a
CSV writer object
        writer.writerow(['Author', 'Text', 'Created
At'])  # Writing the CSV header

        for comment in queryset:  # Iterating over
each comment in the queryset
            writer.writerow([comment.author,
comment.text, comment.created_at])  # Writing comment
data
```

```
        return response  # Returning the response
with CSV data
```

In the code above, we define the `export_as_csv` function that will export comment data into CSV format. This function sets the content type to `text/csv`, writes the CSV header, and then writes each comment into the CSV file.

Adding Custom Action in Admin: Next, we need to add this custom action to the admin interface for the `Comment` model. We can do this by adding the `actions` property in the admin class. Here are the updates to make:

```
# admin.py
class CommentAdmin(admin.ModelAdmin):
    list_display = ('author', 'text', 'created_at')
# Displaying specific columns
    actions = ['export_as_csv']  # Adding custom
action for CSV export

admin.site.register(Comment, CommentAdmin)  #
Registering the Comment model with customization
```

By adding `actions = ['export_as_csv']`, we inform Django Admin to include the CSV export option in the list of actions available in the admin interface.

Run the Server and Test the Export Feature: After saving the changes in the `admin.py` file, run the Django server if it's not already running:

```
$ python manage.py runserver  # Starting the Django
server
```

Now, open the Django Admin interface in your browser and navigate to the `Comment` model. When you select several comments and choose the "Export as CSV" action from the action dropdown, a CSV file will be downloaded to your computer named `comments.csv`. This file will contain all the selected comments with the columns Author, Text, and Created At.

By adding data export features to Django Admin, we provide users with the flexibility to manage and utilize their data as needed. This is particularly useful in situations where data needs to be shared or analyzed further outside the application platform.

3.7 Security In Django Admin

Security is a crucial aspect of web application management, especially when using the admin interface. Django Admin provides powerful tools to manage user access and permissions, which are essential for maintaining data integrity and security. In this section, we will discuss how to manage user permissions in Django Admin, including restricting access based on user groups and permissions that can be configured to ensure that only authorized users can access sensitive data or perform specific actions.

3.7.1 Managing User Perm*issions*

Managing user permissions in Django Admin involves using the permission system provided by Django, which allows us to control who can perform certain actions on models and data.

Django automatically creates permissions for each model we register in the admin, including permissions for reading, adding, editing, and deleting data.

First, we need to understand that each user in Django can be placed in groups called "user groups."

These groups can have the same permissions, making it easier to manage access for a group of users with similar roles. For example, we might have a group for "Editors," who have permission to edit content, and a group for "Viewers," who only have permission to view content.

To manage user permissions, we can do so through the Django Admin interface with the following steps:

After logging into Django Admin, navigate to the "Users" or "Groups" section. Here, we can add or edit users and groups. When adding or editing a user, we will see options to assign permissions directly or through groups. When editing a group, we can select the permissions we want to grant to that group, which will then apply to all users in that group.

For example, if we have an `Article` model, Django will automatically create permissions such as `add_article`, `change_article`, and `delete_article`. We can select these permissions to grant to specific user groups. If we want to restrict a user's access to only reading articles without the ability to edit or delete them, we can set the permissions accordingly when adding the user to the group.

Once permissions are assigned, users will only be able to view and access data according to the permissions granted. This helps ensure that sensitive and important data is not modified or deleted by unauthorized users.

By using this permission system, we not only enhance the security of the application but also keep the data secure and organized. This system is very flexible and allows for customization to meet specific business needs. In the following subsection, we will discuss further steps to restrict access to Django Admin, including how to set up custom URLs for the admin pages to enhance security.

3.7.2 Restricting Access to Django Admin

Restricting access to Django Admin is an important step in maintaining the security of a web application. By doing this, we can prevent unauthorized access and protect sensitive data from unauthorized users. In this subsection, we will explain several steps that can be taken to enhance the security of Django Admin, including the use of custom URLs and user permission settings.

The first step in restricting access is to change the default URL for the admin page. By default, the admin URL in Django can be accessed through `/admin/`. This can be an easy target for attackers, who often try to access the admin page. By changing this URL to something less predictable, we can add an extra layer of protection. To do this, we need to edit the `urls.py` file in our Django project.

For example, we can change the admin URL to `/my-secret-admin/` as follows:

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

urlpatterns = [
    path('my-secret-admin/', admin.site.urls),  # 
Changing admin URL
]
```

By making this change, access to the admin page is now only possible through the new URL we set. This makes it harder for attackers trying to access the admin page to find it.

The next step is to ensure that only users with the appropriate permissions can access Django Admin. As previously discussed, we can manage user and group permissions to restrict access. However, we can also consider using additional authentication, such as two-factor authentication (2FA), to enhance security.

Additionally, we can restrict access to the admin page based on IP address. This can be done by using middleware that checks the user's IP address before allowing access to Django Admin. We can write a simple middleware like the following:

```
# middleware.py
from django.http import HttpResponseForbidden
from django.urls import reverse

class AdminIPRestrictionMiddleware:
    ALLOWED_IPS = ['192.168.1.1']  # Replace with
allowed IP addresses
```

```
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        if
request.path.startswith(reverse('admin:index')) and
request.META['REMOTE_ADDR'] not in self.ALLOWED_IPS:
            return HttpResponseForbidden("You are not
allowed to access this page.")
        return self.get_response(request)
```

In the example above, we create middleware that will check whether the user's IP address is in the allowed list before permitting access to the admin page. If the user's IP address is not in the list, they will receive a "Forbidden" response.

By implementing these measures, we can significantly enhance the security of Django Admin and protect our application from unauthorized access. Security is an ongoing process, and it is essential to continually monitor and update security settings according to the needs of the application. In the following subsection, we will summarize the key points discussed in this chapter and discuss the importance of understanding and maximizing the use of Django Admin.

3.8 Addressing Common Issues In Django Admin

When working with Django Admin, we may encounter some common issues that can hinder access or use of the administration interface. In this section, we will identify frequent problems, such as "Invalid login" and "Superuser cannot access the admin page," and provide explanations and solutions to resolve them.

3.8.1 Common Issues When Accessing Django Admin

One of the most frequent problems users face is difficulty logging into the admin page. The "Invalid login" error message often appears when a user attempts to log in but fails. This issue can typically occur for several reasons:

1. ***Incorrect Credentials***: The user might have entered the wrong username or password. It is important to ensure that the credentials entered are correct and match those created when the superuser was set up. If there's any doubt, we can try resetting the password using the following terminal command:

```
$ python manage.py changepassword [username]
# Change the password for a specific user
```

   By replacing `[username]` with the desired username, we will be prompted to enter a new password.

2. ***Unregistered Superuser***: If the user attempts to log in with an account that is not a superuser, they may not have access to the admin page. Ensure that the superuser has been properly created. If no superuser exists, we need to create one using the command:

```
$ python manage.py createsuperuser  # Create a
new superuser
```

When creating a superuser, be sure to fill in all the required information, including username, email, and password.

3. ***Database Issues***: Sometimes, this issue can be caused by an error in the database. If there are changes to the models that haven't been applied through migration, we need to run the migration commands to ensure all changes are implemented:

```
$ python manage.py migrate  # Apply all
pending migrations
```

If the problem persists after following these steps, we should also check the application's log files to find further clues about the error.

Another common issue is when a superuser is unable to access the admin page at all. This can be caused by several factors, including:

1. ***Restricted Access***: If we have set access restrictions using middleware or other settings, ensure that the superuser has the necessary permissions to access the admin page. We can check the permission settings in Django Admin and ensure the superuser belongs to the group with full access rights.
2. ***URL Configuration Errors***: If the admin URL has been changed and not configured correctly, the superuser might not be able to access it. Ensure that the URL being used matches the settings in `urls.py`.

3. ***Dependency Issues***: Occasionally, this issue may arise from conflicts in the dependencies or packages used in the Django project. Ensure all packages are installed and updated correctly.

By understanding these common issues and the steps to resolve them, we can more quickly find solutions when facing difficulties accessing Django Admin. In the next section, we will discuss further debugging strategies and solutions for more complex issues that may arise during the use of Django Admin.

3.8.2 Debugging and Solutions

After identifying common issues that can hinder access to and use of Django Admin, the next step is to perform debugging to find the root cause of the problem and implement the appropriate solutions. In this subsection, we will discuss several approaches that can be used to address issues such as model errors, permission errors, and other problems frequently encountered in Django Admin.

One common source of problems is errors in the model. If a model is not defined correctly, or if changes are made to the model without applying migrations, this can cause Django Admin to malfunction. To address this, we need to ensure that all changes to the model have been applied. We can do this by running the following commands:

```
$ python manage.py makemigrations  # Create migration
files for model changes
$ python manage.py migrate           # Apply all
pending migrations
```

The first command will create migration files that reflect the changes we have made to the model, while the second command will apply them to the database. If any issues are detected during migration, Django will provide error messages that help us understand what went wrong.

Next, we also need to pay attention to user permission settings. Sometimes, users may not have the necessary permissions to access certain features in Django Admin. To check and manage permissions, we can log in to Django Admin using a superuser account and go to the "Users" and "Groups" sections. Make sure users have permissions that correspond to their roles. If not, we can easily add the required permissions.

If users encounter errors when accessing specific pages, such as "403 Forbidden," this is usually due to permission issues. Ensure that the user is part of a group that has access to the admin page. We can also use User and Group to assign permissions more specifically.

Additionally, if other errors occur while running the server, we can check the error logs for more information. Enabling logging in Django can help with this. In the `settings.py` file, we can add settings for logging:

```
# settings.py
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': 'django_debug.log',  # Log
file
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

With this configuration, all Django logs will be recorded in the `django_debug.log` file. We can check this file to find further clues about the errors that occur.

By following these debugging steps, we can more easily identify and resolve issues that arise when using Django Admin. Knowing how to address these errors not only improves the user experience but also ensures that our application runs smoothly.

Conclusion Of The Chapter

In this chapter, we have explored various important aspects of Django Admin, a powerful tool for content management within Django applications. We began with an introduction to the role and benefits of Django Admin, followed by instructions on how to create a superuser and leverage its key features.

We discussed how to register models in Django Admin and personalize its appearance and functionality to better meet user needs. Features like `list_display`, `list_filter`, and `search_fields` provide greater control over data presentation, while the ability to add custom actions allows us to extend functionality to align with business requirements.

On the security front, we addressed critical aspects such as user permission management and measures to restrict access to the admin interface. This is vital to ensure that only authorized users can access and manage sensitive data.

Equally important, we identified common issues users may encounter while interacting with Django Admin and provided solutions to address them. By understanding how to debug and manage errors, we can ensure the application runs smoothly and maintains a positive user experience.

Overall, this chapter underscores the importance of a deep understanding of Django Admin to maximize the potential of our Django applications. By utilizing the available features and implementing best practices in permission management and security, we can create effective and efficient management systems that support business needs and enhance productivity.

Chapter 4 - Starting the Project

In this chapter, we will set up a Django project for our bot platform. This process involves creating a new project and configuring the folder structure and files needed to begin development.

4.1 Creating A Django Project

We will set up a Django project for our bot platform. This process includes creating a new project and configuring the necessary folder structure and files to start development.
To get started, we need to create a new Django project. Make sure you have Django installed in your development environment.

Once Django is installed, you can create a new project using the following command:

```
$ django-admin startproject platform_bot
```

This command will create a folder named `platform_bot`, which contains the basic files and folders for your Django project. The folder structure will include the `manage.py` file and a `platform_bot` folder that contains the main configuration files.

Inside the `platform_bot` directory, you will find the following project structure:

- ***manage.py***: A script to manage your Django project, such as running the development server, making migrations, and more.
- ***platform_bot/***: A directory that contains the main settings for the Django project.
    - ***init.py***: An empty file indicating that this directory is a Python package.
    - ***settings.py***: A file containing the configuration settings for the Django project.
    - ***urls.py***: A file defining the URL patterns for the Django project.
    - ***wsgi.py***: A file serving as the entry point for the WSGI server.

By following these steps, we have successfully created our first Django project. Next, we will explore Django project settings and begin building a web application with Django.

4.1.1 Creating the `apps` Folder

In Django project development, it is often helpful to organize your apps into separate folders for easier management and development. For the `platform_bot` project, we will create an `apps` folder to store all the apps related to this project. This is a good practice to keep the project structure tidy and well-organized.

Why and How to Create the `apps` Folder

Creating an `apps` folder aims to simplify the project structure and separate different apps in one place. This helps in managing apps more easily, especially as the project grows in complexity.

The first step is to create an `apps` folder in your project directory. To do this, navigate to the `platform_bot` project directory and create a new folder named `apps`:

```
$ mkdir apps
```

Once the `apps` folder is created, we need to move existing apps and add new apps into this folder. By default, the apps you create should be placed in the `apps` folder.

Folder Structure Inside `apps`

Inside the `apps` folder, we will create separate folders for each app required in this project. For `platform_bot`, we will create the following apps:

- ***authentication***: For user registration, login, and logout system.
- ***dashboard***: For the dashboard page where users manage their bots.
- ***users***: For user profiles and profile management.
- ***webhook***: For handling webhooks that interact with the bot.
- ***bots***: For the bot logic and bot management itself.

To create these apps, use the following Django commands inside the `apps` folder:

```
$ cd apps
$ django-admin startapp authentication
$ django-admin startapp dashboard
$ django-admin startapp users
$ django-admin startapp webhook
$ django-admin startapp bots
```

After running these commands, the `apps` folder will have the following structure:

```
platform_bot/
│
├── apps/
│   ├── authentication/
│   ├── dashboard/
│   ├── users/
│   ├── webhook/
│   └── bots/
│
├── platform_bot/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│
├── manage.py
└── db.sqlite3
```

Adding the `apps` Folder to `INSTALLED_APPS`

After creating the apps, you need to ensure that Django is aware of these apps by adding them to the `INSTALLED_APPS` list in the `settings.py` file.

Open `platform_bot/settings.py` and add each app to the `INSTALLED_APPS` list as follows:

```
# File: platform_bot/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'apps.authentication',  # Authentication app
    'apps.dashboard',       # Dashboard app
    'apps.users',           # Users app
    'apps.webhook',         # Webhook app
    'apps.bots',            # Bots app
]
```

By adding the apps to `INSTALLED_APPS`, Django will recognize and load them when the project runs.

With the `apps` folder and the apps created, your project now has a more organized structure, making it easier to develop and maintain the applications in the future. Next, we will dive into the creation and configuration of the applications in more detail.

With this configuration, our Django project is ready for further development. Next, we will add the necessary apps and set up a more detailed folder structure.

4.1.2 Updating `apps.py`

What is `apps.py`?

The `apps.py` file in a Django project is where the configuration for each Django application is defined. Typically, this file contains a class that inherits from `AppConfig`, which is used to set various options and settings related to a specific application.

Why Modify `apps.py`?

By default, Django generates an `apps.py` file with a simplified name for the application. However, when using a more complex project structure, like separating applications into an `apps` folder, it's important to ensure the application's path is correctly defined. If the application path is not properly configured, Django may not be able to locate the app, leading to issues when loading or managing the application.

Below is an example of how to update the `apps.py` file for each application we've created, ensuring that the application path is correctly defined:

Authentication

```
# apps/authentication/apps.py

from django.apps import AppConfig

class AuthenticationConfig(AppConfig):
    default_auto_field =
'django.db.models.BigAutoField'
    name = 'apps.authentication'
```

Webhook

```
# apps/webhook/apps.py

from django.apps import AppConfig
```

```
class WebhookConfig(AppConfig):
    default_auto_field =
'django.db.models.BigAutoField'
    name = 'apps.webhook'
```

Dashboard

```
# apps/dashboard/apps.py

from django.apps import AppConfig

class DashboardConfig(AppConfig):
    default_auto_field =
'django.db.models.BigAutoField'
    name = 'apps.dashboard'
```

Users

```
# apps/users/apps.py

from django.apps import AppConfig

class UsersConfig(AppConfig):
    default_auto_field =
'django.db.models.BigAutoField'
    name = 'apps.users'
```

Bots

```
# apps/bots/apps.py

from django.apps import AppConfig

class BotsConfig(AppConfig):
    default_auto_field =
'django.db.models.BigAutoField'
    name = 'apps.bots'
```

Reasons for Updating `apps.py`:

1. ***Project Structure Consistency:*** If you're using a separate project structure with the `apps` folder, the app paths in `apps.py` must be aligned with this structure so Django can find and load the applications correctly.
2. ***Application Management:*** Defining the correct path ensures that all apps are recognized by Django, making it easier to manage and integrate applications within the project.
3. ***Error Prevention:*** Setting the correct paths helps prevent errors that may occur if Django cannot find a specified app, which could cause issues when running the server or migrating the database.

By updating `apps.py` according to the project structure, we ensure that all applications are properly accessible and managed by Django. If issues persist, it's important to review the configuration and other project code to identify any further potential problems.

4.1.3 Understanding the Basic Structure of an Application

With the `apps` folder and applications created, the next step is to understand the basic structure of each application. This structure includes the model, view, and template that will be used by the application. Additionally, we will configure routing in `urls.py` to link URLs with the appropriate view.

Model

A model in Django is a representation of your application's data. Models define the structure of data to be stored in the database, including data types and relationships between data. Every model in Django is a subclass of `django.db.models.Model` and is usually defined within the `models.py` file.

For example, let's create a basic model for the `authentication` app that handles user information. Open the file `apps/authentication/models.py` and add the following code:

```python
# File: apps/authentication/models.py

from django.db import models
from django.contrib.auth.models import AbstractUser

class CustomUser(AbstractUser):
    # Add additional attributes if needed
    pass
```

Here, `CustomUser` is a subclass of `AbstractUser`, which allows us to extend Django's default user model if needed.

View

In Django, a view is responsible for handling the logic of the application and generating the response that is sent to the user. Views are implemented in the `views.py` file and connect the data from the model with the template.

For example, in the `dashboard` app, we can create a simple view to display the main dashboard page. Open the file `apps/dashboard/views.py` and add the following code:

```python
# File: apps/dashboard/views.py

from django.shortcuts import render

def dashboard_home(request):
    return render(request, 'dashboard/home.html')
```

This view will render the `home.html` template located in the `templates/dashboard/` folder.

Template

A template in Django is an HTML file that defines how data is displayed to the user. Templates are used to render the data sent by the view.

For the `dashboard` app, we will create a basic template in the `templates/dashboard/` folder. Create a new file named `home.html` inside that folder and add the following HTML code:

```html
<!-- File:
apps/dashboard/templates/dashboard/home.html -->

<!DOCTYPE html>
<html>
<head>
    <title>Dashboard Home</title>
</head>
<body>
    <h1>Welcome to Your Dashboard</h1>
```

```
</body>
</html>
```

This template is a basic page that will be displayed when users access the dashboard page.

Configuring Routing in `urls.py`

Routing in Django connects URLs to the appropriate views. You need to configure the `urls.py` file in each app to define the URL patterns and the views that handle those URLs.

Let's start with the `dashboard` app. Open the file `apps/dashboard/urls.py` and add the following configuration:

```
# File: apps/dashboard/urls.py

from django.urls import path
from . import views

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

This configuration connects the root URL of the `dashboard` app to the `dashboard_home` view.

Next, you need to include the `dashboard` app's URLs in the main project routing. Open the file `platform_bot/urls.py` and add the following configuration:

```
# File: platform_bot/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('dashboard/',
include('apps.dashboard.urls')),  # Include the
dashboard app's URLs
]
```

This configuration ensures that any URL starting with `/dashboard/` will be mapped to the `dashboard` app and the `dashboard_home` view.

By setting up the model, view, template, and routing configuration, you have built the basic foundation for your Django application. Next, we will continue adding more functionality and setting up other applications.

4.1.4 Managing Static Files

After understanding the basic structure of an application, the next step is managing static files. Static files, such as CSS, JavaScript, and images, are essential in web application development. Properly managing these files will ensure your application's appearance and functionality meet expectations.

How to Manage Static Files like CSS and JavaScript

Static files are used to control the appearance and interactivity of your application. These include CSS files for styling, JavaScript files for interactive functionality, and images used in various parts of the application. In a Django project, static files are usually grouped within a `static` folder.

For the `platform_bot` project, we will store the static files within the `apps` folder to keep everything in one place. Here are the steps to manage static files:

Create the Static Folder Structure

Inside the `apps` folder, create a subfolder named `static`. Within the `static` folder, you can create additional subfolders for CSS, JavaScript, and images:

```
$ mkdir -p apps/static/
$ mkdir -p apps/static/assets
$ mkdir -p apps/static/assets/css
$ mkdir -p apps/static/assets/js
$ mkdir -p apps/static/assets/img
```

Adding Static Files

Place your CSS, JavaScript, and image files into the appropriate subfolders. For example, add CSS files to the `css` folder, JavaScript files to the `js` folder, and images to the `img` folder.

For instance, create a CSS file named `styles.css` in the `css` folder with the following content:

```
/* File: apps/static/assets/css/styles.css */

body {
    font-family: Arial, sans-serif;
}
```

```
h1 {
    color: #333;
}
```

And create a JavaScript file named `scripts.js` in the `js` folder with the following content:

```
// File: apps/static/assets/js/scripts.js

document.addEventListener('DOMContentLoaded',
function() {
    console.log('JavaScript loaded');
});
```

For images, you can add files such as `logo.png` in the `images` folder.

Static Configuration in Settings

To ensure Django can find and serve static files properly, you need to configure `settings.py`. Open the file `platform_bot/settings.py` and make sure that the `STATIC_URL` and `STATICFILES_DIRS` settings are correctly configured:

```
# File: platform_bot/settings.py

STATIC_URL = '/static/'

STATICFILES_DIRS = [
    BASE_DIR / 'apps/static',
]
```

The `STATIC_URL` setting defines the base URL for static files, while `STATICFILES_DIRS` tells Django where to look for static files outside the standard application folder.

Organizing Static Files within the Project

To ensure static files and templates are well-organized, the expected folder structure is as follows:

```
platform_bot/
│
├── apps/
│   ├── authentication/
│   ├── dashboard/
│   ├── users/
│   ├── webhook/
│   ├── bots/
│   ├── static/
│   │   ├── assets/
│   │           ├── css/
│   │           ├── js/
│   │           └── img/
│
├── platform_bot/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│
├── manage.py
└── db.sqlite3
```

With this configuration, Django will be able to serve static files and templates correctly. Next, we will continue adding additional functionality to the application and ensure all components work as expected.

4.1.5 Organizing Template Files in the `apps` Folder

To ensure that the templates in your Django project are well-structured and easily accessible, we need to follow a few important steps in organizing the template files. In our project, the `templates` folder will be located inside the **`apps`** folder. This ensures that the templates are directly accessible within the relevant application.

Here are the steps to organize template files in a Django project, specifically with a `templates` folder structure inside the **`apps`** folder.

Creating the `templates` Folder

First, create a `templates` folder inside the **`apps`** folder:

```
$ mkdir -p apps/templates/
$ mkdir -p apps/templates/home
$ mkdir -p apps/templates/account
$ mkdir -p apps/templates/dashboard
$ mkdir -p apps/templates/users
$ mkdir -p apps/templates/bots
```

Your project folder structure will look like this:

```
platform_bot/
│
├── apps/
│   ├── authentication/
│   ├── dashboard/
│   ├── users/
│   ├── webhook/
│   ├── bot/
│       ├── static/
│       │   ├── assets/
│       └── templates/
│           ├── home/
│           ├── account/
│           ├── dashboard/
│           ├── users/
│           ├── bots/
│
├── platform_bot/
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│
├── manage.py
└── db.sqlite3
```

Here, the `templates` folder is located within the **`apps`** folder, and inside it, you have subfolders such as `home`, which contain template files like `home.html`. The `templates` folder can also include global template files like `base.html`, which are used across the entire application.

Configuring Templates in `settings.py`

To ensure Django can find your template files, you need to configure the `TEMPLATES` setting in the `settings.py` file. Make sure to add a configuration that points to the `templates` folder inside the `apps` directory.

```
# File: platform_bot/settings.py

import os

# Template configuration
```

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [ BASE_DIR / 'apps/templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [

'django.template.context_processors.debug',

'django.template.context_processors.request',

'django.contrib.auth.context_processors.auth',

'django.contrib.messages.context_processors.messages'
,

'platform_bot.context_processors.cfg_assets_root',  #
Adding a context processor
            ],
        },
    },
]
```

With this configuration, Django will look for templates in the `apps/templates` folder as well as in the application's subfolders, such as `apps/authentication/templates`.

By following these steps, you'll have a well-organized template folder structure, ensuring that all template files are easily accessible and usable in your Django project.

4.1.6 Creating a Context Processor

What is a Context Processor?

A context processor in Django is a feature that allows us to add global data that can be accessed in all templates without having to pass it explicitly through the view. Context processors enable us to define data that will be available in every template, making it easier to manage consistent data throughout the web application.

Function of a Context Processor

The function of a context processor is to provide additional data required by templates across the application automatically. By using a context processor, you don't need to include the same data in every view that renders a template, reducing code duplication and improving the efficiency of managing global data.

Creating a Context Processor File

First, create a new file named `context_processors.py` inside the `apps/` folder. This file will contain the context processor function that retrieves data from the settings and provides it to the template.

Use a text editor to create the file:

```
$ vim context_processors.py
```

Fill the `context_processors.py` file with the following code:

```
# File: apps/context_processors.py

from django.conf import settings

def cfg_assets_root(request):
    # Retrieve the ASSETS_ROOT value from settings.py
and pass it to the template
    return {'ASSETS_ROOT': settings.ASSETS_ROOT}
```

The code above defines the `cfg_assets_root` function, which retrieves the `ASSETS_ROOT` value from the `settings.py` file and returns it as a variable accessible in all templates.

Connecting the Context Processor with Templates

After creating the context processor, the next step is to register it in the `settings.py` file so it can be used in all templates.

Open the `settings.py` file and add the context processor to the `TEMPLATES` configuration:

```
# File: platform_bot/settings.py

# Assets Management
ASSETS_ROOT = '/static/assets'

TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'apps/templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # Django's built-in context
processors

'django.template.context_processors.debug',

'django.template.context_processors.request',
'django.contrib.auth.context_processors.auth',

'django.contrib.messages.context_processors.messages'
,
                # Our custom context processor

'apps.context_processors.cfg_assets_root',  # Adding
the context processor
            ],
        },
    },
]
```

By adding the context processor **`apps.context_processors.cfg_assets_root`** to the **`context_processors`** section, we ensure that the **`ASSETS_ROOT`** variable is available in every template throughout the application without needing to manually add it in every view.

Using Context Processor for Dynamic Data

With the context processor added, we can easily access dynamic data like ASSETS_ROOT in every template. For example, to display the assets directory on the Home page, we can use the variable provided by the context processor without having to add it explicitly in every view.

This step simplifies how we add dynamic elements to web pages, making it easier and more organized to manage global data. With a context processor, we reduce the need to duplicate data in multiple places and ensure that the same data is consistent across all application templates.

4.1.7 Media Settings in `settings.py`

Ensure that the `MEDIA_URL` and `MEDIA_ROOT` settings are correct:

```
# settings.py

import os

# Media files
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

Add Media URL in `urls.py`

Add configuration to serve media files during development:

```
# platform_bot/urls.py
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('apps.authentication.urls')),  #
Include the authentication app URLs
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

4.2 Creating The Home Page (Landing Page)

The home or landing page is the starting point for users when they visit your application. It is the first thing users will see and serves as an introduction to the various features available in the application. In the context of the `platform_bot` project, the home page will act as a gateway for users to register, log in, and get basic information about the application.

4.2.1 Purpose of the Home Page

The home page plays a crucial role in your application. It is where users first interact with the app and is often an essential part of the user experience. This page is typically designed to grab attention and provide clear navigation to the main features of the application. In the `platform_bot` project, the home page will welcome users with a brief introduction to the bot platform and offer options to log in or register.

What Will Be Displayed on the Home Page:

We will display several key elements on the home page:

1. ***Brief Description of the Bot Platform***: Providing basic information about what `platform_bot` is and how it can help users.
2. ***Login and Registration Buttons***: Making it easy for users to sign in or create a new account.
3. ***Links to Additional Information***: Directing users to other pages in the application for more details.

To build this home page, we will focus on the `authentication` app, as it handles user registration and login.

4.2.2 Creating the View for the Home Page

In Django, a view is a component that handles the logic for an HTTP request and returns an appropriate response. For the home page, we will create a view that renders the HTML template we have designed.

A view in Django is a Python function that receives an HTTP request and returns an HTTP response. To create a view in Django, you need to add a view function inside the `views.py` file in the relevant app. In this case, we will add the view in the `authentication` app.

Open the `apps/authentication/views.py` file and add the following function:

```python
# File: apps/authentication/views.py

from django.shortcuts import render

# View for the home page
def home(request):
    context = {
        'ASSETS_ROOT': '/static/assets',  # 
Additional path for static assets if needed
    }
    return render(request, 'home/home.html', context)
```

In this example, the `'ASSETS_ROOT'` context is passed to the template, allowing you to use this variable in the template to manage static files more dynamically.

The `home` function above uses `render` to render the `home.html` template. The `render` function makes it easy to connect data from the view to the HTML template displayed to the user.

Explanation of `render` and Data Context

The `render` function is a highly useful tool in Django. It combines an HTML template with the provided context data and returns an `HttpResponse` that can be sent to the user.

The syntax for the `render` function is as follows:

```
render(request, template_name, context=None)
```

- ***request***: The HTTP request object.
- ***template_name***: The name of the HTML template file to render.
- ***context***: (Optional) A dictionary containing the data you want to include in the template.

In the HTML template, you can use data from the context like this:

```
<!-- File: apps/templates/authentication/base.html --
>
```

```
<link href="{{ ASSETS_ROOT }}/css/style.css"
rel="stylesheet"/>
```

This view will render the `home.html` template, which we will create in the next step.

In the example above, the `ASSETS_ROOT` variable is now available in the template, and we use it to define the path to images or other static files. This makes it easier to manage asset paths dynamically, especially if assets are stored in a specific location that may change.

4.2.3 Creating a Template for the Home Page

Template inheritance is a feature in Django that allows you to create a base template that can be extended by other templates. This is very useful for maintaining consistent layouts across your entire application.

Next, we will create template files for the home page. Open the folder named `home` inside the `apps/templates` folder. In this folder, create two files named `base.html` and `home.html`:

```
$ touch apps/templates/home/base.html
$ touch apps/templates/home/home.html
```

For example, you can create a basic template called `base.html` in the `apps/templates` folder that contains common elements like the header and footer:

```
<!-- File: apps/templates/home/base.html -->
```

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Platform Bot{% endblock
%}</title>
</head>
<body>
    <header>
        <h1>Platform Bot</h1>
        <nav>
            <a href="{% url 'home' %}">Home</a>
            <a href="#">Login</a>
            <a href="#">Register</a>
        </nav>
    </header>
    <main>
        {% block content %}
        <!-- Page-specific content will go here -->
        {% endblock %}
    </main>
    <footer>
        <p>&copy; 2024 Platform Bot. All rights
reserved.</p>
    </footer>
</body>
</html>
```

Then, in the `home.html` template, you can extend from `base.html` and fill in the content blocks that were defined:

```
<!-- File: apps/templates/home/home.html -->

{% extends 'home/base.html' %}

{% block title %}Home - Platform Bot{% endblock %}

{% block content %}
    <h2>Welcome to Platform Bot</h2>
    <p>Your gateway to managing and interacting with
your bots.</p>
{% endblock %}
```

By using template inheritance, you can keep your HTML templates more organized and make your application easier to maintain.

This template will display a welcome message, a brief description, and links to the login and registration pages.

4.2.4 Structuring the Home Page Layout

After successfully creating the template and view for the home page, the next step is to organize the layout to look attractive, modern, and responsive across various devices, including smartphones and tablets. Responsive layout is essential because it allows the webpage to adjust its appearance based on the user's screen size.

Explanation of Responsive Layout

Responsive layout is a web design technique where the elements on the page dynamically adjust their position and size according to the screen width. This is important because users access web applications from various devices with different resolutions, such as mobile phones, tablets, laptops, or desktops.

To create a responsive layout, we need to use flexible CSS rules, such as adaptive units of measurement (e.g., `em`, `rem`, or `%`), as well as a grid system or CSS framework designed specifically for responsive design, such as Bootstrap.

Using CSS and Frameworks like Bootstrap

One of the most efficient ways to create a responsive layout is by using a CSS framework like Bootstrap. Bootstrap provides a flexible grid system and components that make it easy to create responsive and interactive page layouts.

The first step is to add the Bootstrap files to the project. You can either add the Bootstrap CSS file directly via a CDN or download it and save it in the `static` folder.

Adding Bootstrap via CDN

You can add Bootstrap to the home page by including the Bootstrap CDN link in the `<head>` section of the `base.html` template.

```
<!-- File: apps/templates/home/base.html -->

<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Platform Bot{% endblock
%}</title>
    <!-- Adding Bootstrap from CDN -->
  <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhj
Y6hW+ALEwIH" crossorigin="anonymous">
</head>
<body>
    <header class="bg-primary text-white text-center
py-3">
        <h1>Platform Bot</h1>
        <nav>
            <a href="{% url 'home' %}" class="text-
white">Home</a>
```

```
            <a href="#" class="text-white">Login</a>
            <a href="#" class="text-
white">Register</a>
        </nav>
    </header>
    <main class="container mt-5">
        {% block content %}
        {% endblock %}
    </main>
    <footer class="text-center py-4 bg-dark text-
white">
        <p>&copy; 2024 Platform Bot. All rights
reserved.</p>
    </footer>
</body>
</html>
```

By adding the above code, you have set up Bootstrap for use across all your web pages. Next, we will apply the Bootstrap grid system to the `home.html` page.

Structuring the Home Page Layout with Bootstrap

Now we will add the grid system to the home page to make it organized and responsive. In the `home.html` template, we can use Bootstrap classes to create a grid structure.

```
<!-- File: apps/templates/home/home.html -->

{% extends 'home/base.html' %}

{% block title %}Home - Platform Bot{% endblock %}

{% block content %}
<div class="jumbotron text-center">
    <h1 class="display-4">Welcome to Platform
Bot</h1>
    <p class="lead">Your gateway to managing and
interacting with your bots.</p>
```

```
    <hr class="my-4">
    <p>Create and manage your bots easily through our
platform.</p>
    <a class="btn btn-primary btn-lg" href="#"
role="button">Get Started</a>
</div>

<div class="row">
    <div class="col-md-4">
        <h3>Easy to Use</h3>
        <p>Our platform is designed to be simple and
intuitive, allowing you to easily create and manage
bots.</p>
    </div>
    <div class="col-md-4">
        <h3>Flexible</h3>
        <p>Customize your bots according to your
needs with our flexible settings and tools.</p>
    </div>
    <div class="col-md-4">
        <h3>Scalable</h3>
        <p>Whether you manage one bot or hundreds,
our platform can scale to meet your business
needs.</p>
    </div>
</div>
{% endblock %}
```

In the layout above:

- ***Jumbotron***: Bootstrap provides a jumbotron component, which is a large, attention-grabbing element often used for the main message or highlight of the page.
- ***Grid System***: We use `row` and `col-md-4` to create a responsive three-column grid. Each column will adjust its size based on the user’s screen width. On smaller screens, the columns will stack vertically, while on larger screens, they will appear in a single row.

Using Custom CSS (Optional)

In addition to Bootstrap, you can add custom styles through your own CSS file. Create a file called `styles.css` in the `static/css` folder and add your custom styles there. Here's a simple example:

```css
/* File: apps/static/assets/css/styles.css */

body {
    background-color: #f8f9fa;
}

header {
    background-color: #007bff;
    color: white;
    padding: 10px;
}

footer {
    background-color: #343a40;
    color: white;
    padding: 20px;
}

.jumbotron {
    background-color: #e9ecef;
    padding: 2rem 1rem;
}
```

Make sure to include this file in the `base.html` template so it applies across all pages:

```html
<link href="{{ ASSETS_ROOT }}/css/style.css"
rel="stylesheet"/>
```

With these steps, you have successfully structured the home page layout using Bootstrap and custom CSS. This layout is not only visually appealing but also flexible and accessible from various devices, ensuring an optimal user experience.

4.2.5 Adding Navigation and Interactive Elements

After laying out the basic design for the home page, the next step is to add navigation and interactive elements so that users can easily perform actions like logging in and registering. Good navigation should be intuitive and easily accessible from any device. Additionally, interactive elements such as buttons and forms will help users navigate the page and interact with the application.

Creating Navigation for Login and Registration
Navigation is one of the most important elements on the home page. In this project, we will create navigation that includes two main options: Login and Registration. These two elements will be positioned in the top right corner of the page to ensure easy accessibility for users.

This navigation can be created using the Navbar component from Bootstrap, which we have already added previously. Here is an example of adding navigation in the file `apps/templates/authentication/base.html`:

```
<!-- File: apps/templates/home/base.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Bot Platform</title>
    <!-- Linking Bootstrap CSS from CDN -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhj
Y6hW+ALEwIH" crossorigin="anonymous">
</head>
<body>
    <!-- Header section for navigation -->
    <header>
        <nav class="navbar navbar-expand-lg navbar-
light bg-light">
            <a class="navbar-brand" href="#">Bot
Platform</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 ms-auto">
                    <li class="nav-item">
                        <a class="nav-link"
href="#">Login</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link"
href="#">Register</a>
                    </li>
                </ul>
            </div>
        </nav>
    </header>
```

Code Explanation:

1. ***Navbar:*** This navigation uses the navbar element from Bootstrap, which has been set to be responsive. If users

open the page on a mobile device, this menu will change to a dropdown button to save space.

2. ***Login and Register:*** The two links in the navigation will direct to the login page (`{% url 'login' %}`) and registration page (`{% url 'register' %}`). These links make it easy for users to take action right after viewing the home page. However, since we have not yet created the pages and views for login and registration, we have not added them.

Adding JavaScript for Additional Interaction

If you want to add further interactive elements, such as form validation or transitions, Bootstrap also provides a JavaScript library that you can use. We have already included this library at the bottom of the previous page:

```
<!-- Including Bootstrap JavaScript -->
<script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11
.8/dist/umd/popper.min.js" integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc
2pM8ODewa9r" crossorigin="anonymous"></script>
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/js/bootstrap.min.js" integrity="sha384-
0pUGZvbkm6XF6gxjEnlmuGrJXVbNuzT9qBBavbLwCsOGabYfZo0T0
to5eqruptLy" crossorigin="anonymous"></script>
```

This allows us to add dynamic interactions on the page, such as dropdown animations, modals, and more, without writing a lot of manual JavaScript code. You can easily activate these interactive components by adding the appropriate HTML attributes to the desired elements.

Complete Code

```
<!-- File: apps/templates/home/base.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Bot Platform</title>
    <!-- Linking Bootstrap CSS from CDN -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhj
Y6hW+ALEwIH" crossorigin="anonymous">
</head>
<body>
    <!-- Header section for navigation -->
    <header>
        <nav class="navbar navbar-expand-lg navbar-
light bg-light">
            <a class="navbar-brand" href="#">Bot
Platform</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 ms-auto">
                    <li class="nav-item">
                        <a class="nav-link"
href="#">Login</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link"
href="#">Register</a>
                    </li>
                </ul>
            </div>
        </nav>
    </header>
    <main class="container mt-5">
        {% block content %}
        {% endblock %}
    </main>
    <footer class="text-center py-4 bg-dark text-
white">
        <p>&copy; 2024 Bot Platform. All rights
reserved.</p>
    </footer>
    <!-- Including Bootstrap JavaScript -->
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11
.8/dist/umd/popper.min.js" integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc
2pM8ODewa9r" crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/js/bootstrap.min.js" integrity="sha384-
0pUGZvbkm6XF6gxjEnlmuGrJXVbNuzT9qBBavbLwCsOGabYfZo0T0
to5eqruptLy" crossorigin="anonymous"></script>
</body>
</html>
```

By arranging navigation and interactive elements this way, the home page becomes more functional and accessible to users.

4.2.6 Configuring URLs for the Home Page

Next, we need to add URL configurations for the home page. Open the file `apps/authentication/urls.py` and add a new URL for the home view:

```
# File: apps/authentication/urls.py

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

```
urlpatterns = [
    path('', home, name='home'),
    # Add URLs for login and registration here
]
```

Next, we need to ensure that the authentication app's URL is included in the main project routing. Open the file `platform_bot/urls.py` and add the following configuration:

```
# File: platform_bot/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('apps.authentication.urls')),  # 
Including the authentication app's URLs
]
```

With this configuration, the home page will be displayed when users visit the root URL of the application.

By following the steps above, you have successfully created a home page (landing page) for the `platform_bot` application. This page will welcome users with basic information and provide options for login or registration. Next, we will continue by adding further functionality and ensuring the application works as expected.

4.2.7 Testing the Home Page

After all configurations are complete, the last step is to test whether the home page functions properly.

Running the Server

First, make sure the Django server is running. If not, start the server using the following command:

```
$ python manage.py runserver
```

Accessing the Home Page

Open your browser and access the home page by visiting the following URL:

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

If all configurations have been done correctly, you should see the home page styled with Bootstrap. Ensure that all elements are displayed correctly, including the header, hero section, features section, and buttons.

With this, your home page is complete, including the view, template, URL routing, styling, and static assets. If there are any issues, be sure to double-check the steps above and ensure that all files and configurations have been done correctly.

Conclusion Of The Chapter

In this chapter, we have learned the steps to build a landing page (home page) for the Platform Bot project. This page serves as the main entrance that welcomes new users and guides them to register or log into the platform.

We began by understanding the purpose and role of the home page in the application. Then, we discussed how to create a view for the home page using Django and integrated it with an HTML template that utilizes template inheritance to keep the code structured and organized.

Next, we managed a responsive layout using a CSS framework like Bootstrap, ensuring that the page view can adapt to various devices. Finally, we added navigation and interactive elements such as buttons and links to the login and registration pages to enhance the user experience.

With the home page built, our platform now has a solid foundation as a user interface. The next step is to develop additional features like the dashboard page and bot management, which we will discuss in the following chapters.

All these steps ensure that we have a strong foundation to begin developing web applications using Django.

Chapter 5 - Registration, Login, and Logout Features

In this section, we will discuss and build features for user registration, login, and logout. The registration form is an essential part of an application that allows users to sign up with the required details. We will also add a login and logout system for users in the Django application. This process involves creating a login form, views to handle login, and templates to display the login form to users.

5.1 Building The Registration Feature

We will cover how to build a registration feature for users in your Django application. This registration feature allows new users to create accounts on your platform. We will also discuss the use of user models, including both Django's built-in user model and custom user models if needed.

5.1.1 Creating a User Model

Django provides a built-in `User` model that can be used for user authentication. However, if you need additional information about the users, you can create an extra user profile model that connects to the `User` model. In this example, we will use Django's built-in `User` model and create a `UserProfile` model to store additional information about the users.

The `UserProfile` model will store additional information such as city, address, country, postal code, a description about the user, company name, and profile picture. This model uses a one-to-one relationship with the `User` model, allowing each user to have one additional profile.

Here is the code for the `UserProfile` model:

```
# File: apps/authentication/models.py

from django.contrib.auth.models import User
from django.db import models

class UserProfile(models.Model):
    user = models.OneToOneField(User,
on_delete=models.CASCADE)
    city = models.CharField(max_length=100,
blank=True)
    address = models.CharField(max_length=255,
blank=True)
    country = models.CharField(max_length=100,
blank=True)
    postal_code = models.CharField(max_length=20,
blank=True)
    about = models.TextField(blank=True)
    company = models.CharField(max_length=100,
blank=True)
    foto_profile =
models.ImageField(upload_to='profile_pics/',
blank=True)

    def __str__(self):
        return self.user.username
```

Model Explanation:

- ***user***: One-to-one relationship with the `User` model, meaning each `UserProfile` entry is linked to one `User` entry.
- ***city***: Stores the user's city, optional.
- ***address***: Stores the user's address, optional.
- ***country***: Stores the user's country, optional.
- ***postal_code***: Stores the user's postal code, optional.
- ***about***: Stores additional information or a description about the user, optional.
- ***company***: Stores the user's company name, optional.
- ***foto_profile***: Stores the user's profile picture, uploaded to the `profile_pics/` folder.

Next Steps

Once the `UserProfile` model is created, we need to inform Django to create the corresponding table in the database. This process is done by performing migrations.

The first step is to create a migration file by running the following command in the terminal:

```
$ python manage.py makemigrations
```

Successful output: If the command is successful, you will see a message like this:

```
Migrations for 'users':
  users/migrations/0001_initial.py
    - Create model UserProfile
```

This command generates a migration file for the `users` app. Once the migration file is created, run the following command to apply the migration and create the table in the database:

```
$ python manage.py migrate
```

Successful output: If the command is successful, you will see a message like this:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes,
sessions, users
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying
admin.0003_logentry_add_action_flag_choices... OK
  Applying
contenttypes.0002_remove_content_type_name... OK
  Applying
auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length...
OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying
auth.0007_alter_validators_add_error_messages... OK
  Applying
auth.0008_alter_user_username_max_length... OK
  Applying
auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length...
OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying
auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
  Applying users.0001_initial... OK
```

Once the migration is complete, the new table for the `UserProfile` model will be created in your database.

The `makemigrations` command creates a migration file based on the changes made to the model, while the `migrate` command applies the migration to the database.

With this model, you can store additional information about users that is not provided by Django’s built-in `User` model. This gives you flexibility to customize user profiles according to your application's needs.

Adding the `UserProfile` Model to Django Admin

To manage the `UserProfile` model through the Django admin site, we need to register it in the `admin.py` file. This allows the admin to view, add, and edit user profiles from the admin page.

Open the `authentication/admin.py` file and add the following code:

```
# File: apps/authentication/admin.py

from django.contrib import admin
from .models import UserProfile

admin.site.register(UserProfile)
```

The code above registers the `UserProfile` model on Django's admin site, making it manageable from the admin interface.

Now, if you run the Django server and open the admin site, you will see the `UserProfile` model there. You can add, edit, or delete user profiles through this interface.

5.1.2 Creating a Registration Form

In this section, we will discuss how to create a user registration form using Django. This form will allow users to register on your application by filling in the necessary information.

To create the registration form, we will use the `forms.py` in the authentication app. This form will collect data from the user and perform validation to ensure that the input is as expected.

Here is the code for the registration form using Django's `UserCreationForm` class, with additional fields for user information:

```
# File: apps/authentication/forms.py

from django import forms
from django.contrib.auth.forms import
UserCreationForm
from django.contrib.auth.models import User

class UserRegistrationForm(UserCreationForm):
    email =
forms.EmailField(widget=forms.EmailInput(attrs={'clas
s': 'form-control', 'placeholder': 'Email'}))
```

```
    username =
forms.CharField(widget=forms.TextInput(attrs={'class'
: 'form-control', 'placeholder': 'Username'}))
    password1 =
forms.CharField(widget=forms.PasswordInput(attrs={'cl
ass': 'form-control', 'placeholder': 'Password'}))
    password2 =
forms.CharField(widget=forms.PasswordInput(attrs={'cl
ass': 'form-control', 'placeholder': 'Confirm
Password'}))
    first_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'First Name'}))
    last_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Last Name'}))
    company = forms.CharField(max_length=100,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Company'}))
    address = forms.CharField(max_length=255,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Address'}))
    city = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'City'}))
    country = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Country'}))
    postal_code = forms.CharField(max_length=20,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Postal Code'}))
    about =
forms.CharField(widget=forms.Textarea(attrs={'class':
'form-control', 'placeholder': 'About me'}),
required=False)
    foto_profile = forms.ImageField(required=False,
widget=forms.FileInput(attrs={'class': 'form-control-
file'}))

    class Meta:
        model = User
        fields = ['username', 'email', 'password1',
'password2', 'first_name', 'last_name', 'company',
'address', 'city', 'country', 'postal_code', 'about',
'foto_profile']
```

Form Explanation:

- `email`: A field for the user's email, with an `EmailInput` widget that adds a placeholder and CSS class.
- `username`: A field for the username using a `TextInput` widget.
- `password1` and `password2`: Fields for passwords, using `PasswordInput` widgets to hide the user's input.
- `first_name` and `last_name`: Fields for the user's first and last names.
- `company`: An optional field for the user's company name.
- `address`: A field for the user's address.
- `city`: A field for the user's city.
- `country`: A field for the user's country.
- `postal_code`: A field for the user's postal code.
- `about`: An optional field for additional information about the user.
- `foto_profile`: An optional field to upload a profile picture.

User Input Validation: Django provides built-in validation for forms, including validation for emails and passwords. You can add additional validation if needed by defining a `clean` method in the form. For example, if you want to validate that the two entered passwords match, you can use this method to check that condition.

With this registration form, users can sign up with the required information, and the form will ensure that the data is entered in the correct format. Make sure to add this form to the corresponding views and templates to complete the registration feature.

5.1.3 Creating the Registration View

After creating the registration form, the next step is to handle the registration request in a view. In this section, we will process the input from the form, validate it, and save the user's data into the database. Additionally, once registration is successful, the user will be redirected to a specific page, such as a dashboard or the homepage.

The view is responsible for handling requests from the user. In this case, we will handle a `POST` request from the registration form, validate the user's input, save the user's data into the `User` model, and create a `UserProfile` associated with the user. Additionally, the user will be logged in automatically after successful registration and redirected to the appropriate page.

Here is the code for the registration view:

```
# File: apps/authentication/views.py
```

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from .forms import UserRegistrationForm
from .models import UserProfile

# View to handle new user registration
def user_register(request):
    if request.method == 'POST':
        # Check if the request is POST and process
the form data
        form = UserRegistrationForm(request.POST,
request.FILES)
        if form.is_valid():
            # Save user data into the User model
            user = form.save()
            # Create and save a user profile
associated with the User
            UserProfile.objects.create(
                user=user,
                city=form.cleaned_data.get('city',
''),

address=form.cleaned_data.get('address', ''),

country=form.cleaned_data.get('country', ''),

postal_code=form.cleaned_data.get('postal_code', ''),
                about=form.cleaned_data.get('about',
''),

foto_profile=request.FILES.get('foto_profile', None)
            )
            # Automatically log in the user after
registration
            login(request, user)
            # Redirect to the home or dashboard page
after successful registration
            return redirect('home')
    else:
        # If the request is not POST, create an empty
form to display on the page
        form = UserRegistrationForm()
    # Prepare context for the template
    context = {
        'form': form,
        'ASSETS_ROOT': '/static/assets',  # Include 
the path to assets
    }

    # Render the register page with the prepared form
    return render(request, 'account/register.html', 
context)
```

Code Explanation:

1. ***Request Handling***: This view first checks if the received request is a `POST`. If it's a `POST`, it means the user has submitted the registration form.
2. ***Form Validation***: The form submitted by the user is validated using `form.is_valid()`. If valid, the data is saved to the `User` model.
3. ***UserProfile***: After the user is successfully registered, the user's profile (model `UserProfile`) is also created and saved. Additional data such as city, address, and `foto_profile` are taken from the form and linked to the newly registered user.
4. ***Automatic Login***: After the user is successfully registered, we automatically log them in using Django's `login()` function, so the user doesn't need to log in manually after registration.
5. ***Redirect After Registration***: Once the registration and login are successful, the user is redirected to the homepage or another specified page using `redirect('home')`.

6. ***Form Rendering***: If the request is not a `POST` (e.g., `GET`), an empty form is created and displayed on the registration page (`register.html`).

By creating this view, we have completed the registration request handling. Users will be able to sign up, the system will validate the entered data, and if valid, users will be directly redirected to their main page or dashboard.

5.1.4 Creating the Registration Template

After setting up the registration view, the next step is to create an HTML template for the registration form. This template will display the form to users, allowing them to input the necessary information and showing validation messages if there are any input errors. We will also add styling to make the form visually appealing and in line with the desired design.

To create the registration template, we will create an HTML file in the `templates` folder within the `authentication` application folder. Here are the steps to create the registration template:

Creating the HTML Template

Create an HTML template file in the `templates/account/` folder named `register.html`. This template will display the registration form we created earlier.

```
<!-- File: apps/templates/account/register.html -->
```

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

{% block title %}Register - Bot Platform{% endblock
%}

{% block content %}
<div class="container mt-5">
    <h2 class="text-center">New User
Registration</h2>
    <form method="POST" enctype="multipart/form-
data">
        {% csrf_token %}
        <div class="form-group">
            {{ form.username.label_tag }}
            {{ form.username }}
            {% if form.username.errors %}
                <div class="text-
danger">{{ form.username.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.email.label_tag }}
            {{ form.email }}
            {% if form.email.errors %}
                <div class="text-
danger">{{ form.email.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.password1.label_tag }}
            {{ form.password1 }}
            {% if form.password1.errors %}
                <div class="text-
danger">{{ form.password1.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.password2.label_tag }}
            {{ form.password2 }}
            {% if form.password2.errors %}
                <div class="text-
danger">{{ form.password2.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.first_name.label_tag }}
            {{ form.first_name }}
            {% if form.first_name.errors %}
                <div class="text-
danger">{{ form.first_name.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.last_name.label_tag }}
            {{ form.last_name }}
            {% if form.last_name.errors %}
                <div class="text-
danger">{{ form.last_name.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.company.label_tag }}
            {{ form.company }}
            {% if form.company.errors %}
                <div class="text-
danger">{{ form.company.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.address.label_tag }}
            {{ form.address }}
            {% if form.address.errors %}
                <div class="text-
danger">{{ form.address.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.city.label_tag }}
            {{ form.city }}
            {% if form.city.errors %}
                <div class="text-
danger">{{ form.city.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.country.label_tag }}
            {{ form.country }}
            {% if form.country.errors %}
                <div class="text-
danger">{{ form.country.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.postal_code.label_tag }}
            {{ form.postal_code }}
            {% if form.postal_code.errors %}
                <div class="text-
danger">{{ form.postal_code.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.about.label_tag }}
            {{ form.about }}
            {% if form.about.errors %}
                <div class="text-
danger">{{ form.about.errors }}</div>
            {% endif %}
        </div>
        <div class="form-group">
            {{ form.foto_profile.label_tag }}
            {{ form.foto_profile }}
            {% if form.foto_profile.errors %}
                <div class="text-
danger">{{ form.foto_profile.errors }}</div>
            {% endif %}
        </div>
        <button type="submit" class="btn btn-
primary">Register</button>
    </form>
</div>
{% endblock %}
```

Template Explanation:

1. ***Template Inheritance***: The `register.html` template extends the base template (`base.html`). This ensures that the registration form uses the same layout and styling as the other pages.

2. ***Form Rendering***: The form is rendered using the `form.field_name` method, which is generated from the `UserRegistrationForm`. For each field, we display the label, input field, and if there are any errors, we display the relevant error messages.
3. ***CSRF Token***: The CSRF token is included using `{% csrf_token %}` to protect the form from Cross-Site Request Forgery (CSRF) attacks.
4. ***Styling and Validation***: The form uses Bootstrap classes for styling. Error messages for each field are displayed below the input field if there are validation errors, helping users understand what went wrong with their input and how to correct it.
5. ***Enctype***: The `enctype="multipart/form-data"` attribute is required on the form tag for file uploads, such as profile pictures.

With this template, users will be able to see a registration form with consistent styling and fill in their data immediately. Input validation messages are also clearly displayed, helping users fix errors before submitting the form.

5.1.5 Adding Registration URL Routing

After completing the setup of the model, form, view, and template for registration, the next step is to add URL routing for this registration page.

Add the following path in the `urls.py` file of the `authentications` application:

```
# File: apps/authentications/urls.py

from django.urls import path
from .views import user_register

urlpatterns = [
    path('register/', user_register,
name='register'),
]
```

Run your local server using the following command:

```
$ python manage.py runserver
```

After that, open your browser and access the registration page. Try logging in with both correct and incorrect credentials to test the validity of the authentication system and error message handling.

5.1.6 Testing the Registration Feature (Optional)

After completing the setup of the model, form, view, and template for registration, the final and very important step is to test the functionality of this registration feature. Testing ensures that this feature works as expected and helps avoid bugs or errors in the future. These steps include testing form validation, data submission, user creation, and checking if users can log in automatically after registration.

In this testing phase, we will use the Django Testing Framework to test various aspects of the registration feature.

Creating Test Cases for Registration

First, we will create a test file for the authentication application. These tests will be in the `tests.py` file within the application folder.

Create or open the `tests.py` file:

```
# File: apps/authentication/tests.py

from django.test import TestCase
from django.urls import reverse
from django.contrib.auth.models import User
from apps.authentications.forms import
UserRegistrationForm

class UserRegistrationTest(TestCase):
    # Setup for initial tests
    def setUp(self):
        self.register_url = reverse('register')  #
URL for the registration page
        self.user_data = {
            'username': 'testuser',
            'email': 'testuser@example.com',
            'password1': 'testpassword123',
            'password2': 'testpassword123',
            'first_name': 'Test',
            'last_name': 'User',
            'city': 'Test City',
            'country': 'Test Country',
            'postal_code': '12345',
            'about': 'This is a test user.',
            'company': 'Test Company',
            'address': '123 Test Street'
        }

    # Test if the registration page is accessible
    def test_register_page_accessible(self):
        response = self.client.get(self.register_url)
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response,
'home/register.html')

    # Test if the registration form is valid with
correct data
    def test_registration_form_valid(self):
        form =
UserRegistrationForm(data=self.user_data)
        self.assertTrue(form.is_valid())

    # Test if a new user is created after
registration
    def test_user_created_after_registration(self):
        response =
self.client.post(self.register_url, self.user_data)
        self.assertEqual(response.status_code, 302)
# Should redirect after successful registration

self.assertTrue(User.objects.filter(username='testuse
r').exists())

    # Test validation if passwords do not match
    def test_registration_password_mismatch(self):
        self.user_data['password2'] =
'wrongpassword123'
        form =
UserRegistrationForm(data=self.user_data)
        self.assertFalse(form.is_valid())
        self.assertIn('password2', form.errors)

    # Test if the user is logged in automatically
after registration
    def test_user_login_after_registration(self):
        response =
self.client.post(self.register_url, self.user_data,
follow=True)

self.assertTrue(response.context['user'].is_authentic
ated)
```

Running the Tests

After writing the test cases, we can run the tests to ensure that the registration feature works well. The tests will include several scenarios: accessing the registration page, testing the form, validating passwords, and checking if users can log in immediately after successful registration.

To run the tests, open a terminal and navigate to your Django project directory, then execute the following command:

```
$ python manage.py test apps.authentications
```

If the tests run successfully, you will see output similar to the following:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------------------------
Ran 4 tests in 0.543s

OK
```

Explanation of Tests:

- ***test_register_page_accessible***: Tests if the registration page can be accessed with a status code of 200 and whether the correct template is used.
- ***test_registration_form_valid***: Tests the validity of the registration form with valid data. The form should be valid if all provided data is correct.
- ***test_user_created_after_registration***: Tests if a new user is created after the data is submitted to the server via the registration form. Upon success, the new user will be

saved in the database, which is checked using the filter method.

- ***test_registration_password_mismatch***: Tests error validation if the user enters two mismatching passwords. This test ensures that Django returns an error on the form if the passwords do not match.
- ***test_user_login_after_registration***: Tests if the user is logged in automatically after successful registration. We check if the view context contains the authenticated user object.

By conducting these tests, we ensure that the registration feature functions correctly, from page access, input validation, to automatic login. Testing is an important step in the application development process as it helps prevent future issues and ensures the application works as expected.

5.2 Creating The Login Feature

After successfully creating the registration feature, the next step is to create the login feature. This feature is essential to allow registered users to access their accounts. We will use `forms.py` to create a login form that enables users to enter their credentials (username and password) and authenticate them to log into the system.

5.2.1 Creating the Login Form

To create the login form, we will use Django's built-in class `AuthenticationForm`. This class provides a basic form for user authentication, where we can customize it by adding UI designs to the fields, such as adding placeholders and CSS classes.

Open the `forms.py` file located inside the authentication app folder and add the following code. The form consists of two main fields: username and password, with added CSS attributes for a better user interface.

```
# File: apps/authentications/forms.py

from django.contrib.auth.forms import
UserCreationForm, AuthenticationForm

# User login form
class UserLoginForm(AuthenticationForm):
    # Field for username
    username = forms.CharField(
        widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Username'})
    )
    # Field for password
    password = forms.CharField(
        widget=forms.PasswordInput(attrs={'class':
'form-control', 'placeholder': 'Password'})
    )
```

Explanation of the form:

- ***username:*** This field uses the `TextInput` widget with added class attributes for styling and a placeholder to guide the user.
- ***password:*** This field uses the `PasswordInput` widget, which automatically hides the text entered by the user.

This form will be used to capture input from users when they attempt to log in to the application.

User Validation and Authentication

Once the form is created, Django will handle the user authentication process through the `AuthenticationForm`. This form ensures that the combination of the entered username and password matches the data stored in the database. If validation is successful, the user will be allowed to log in. If an error occurs (such as an incorrect password or the user not being found), an error message will be returned to the form.

Further customization of the authentication process can be done by adding additional logic in the view or middleware. However, for now, we will use Django's default mechanism, which is sufficient.

5.2.2 Creating the View for Login

After creating a functioning login form to capture user input, the next step is to create a view that will handle the login process itself. This view is responsible for validating user credentials and managing the authentication logic using the form we have created. If the login is successful, the user will be redirected to the appropriate page, such as a dashboard. However, if the login fails, we also need to display informative error messages.

Handling Login and Redirecting Users

To handle the login process, we can utilize Django's view. The `user_login` function will be responsible for processing the input from the login form, validating credentials, and authenticating the user.

Here is the implementation of the login view, which should be placed in the `views.py` file in the authentication app:

```
# File: apps/authentications/views.py

from django.contrib.auth import authenticate, login
from django.contrib import messages
from .forms import UserRegistrationForm,
UserLoginForm

# View for user login
def user_login(request):
    if request.method == 'POST':
        # Retrieve form data and validate it
        form = UserLoginForm(data=request.POST)
        if form.is_valid():
            # Authenticate user
            user =
authenticate(username=form.cleaned_data['username'],
password=form.cleaned_data['password'])
            if user is not None:
                # Log in the user if credentials are
valid
                login(request, user)
                messages.success(request, 'Login
successful.')
                return redirect('home')  # Redirect
the user after successful login
            else:
                # Display error message if
authentication fails
                messages.error(request, 'Invalid
username or password.')
    else:
        form = UserLoginForm()

    context = {
        'ASSETS_ROOT': '/static/assets',  #
Additional path for static assets if needed
        'form': form,
    }
    return render(request, 'account/login.html',
context)
```

Code Explanation:

- ***POST Request Processing:*** When the user submits the login form (with the POST method), this view will receive the form data and validate the input using `UserLoginForm`.
- ***User Authentication:*** If the form is valid, the authentication process starts using the `authenticate()` function, which is a built-in Django function that checks whether the entered username and password combination is correct. If authentication is successful, the user will be logged in using the `login()` function and then redirected to the home page.
- ***User Feedback:*** If the login is successful, we display a success message using `messages.success()`. However, if authentication fails (e.g., due to incorrect username or password), we display an error message using `messages.error()`.
- ***GET Request:*** If the page is accessed using the GET method (such as when the user first opens the login page), the login form will be displayed by default.

Adding Feedback for Failed Login Attempts

Django provides a messaging framework to give feedback to users in various contexts. In this login view, we use the `messages` function to display a message when login fails. If the combination of username and password does not match, the user will see an error message indicating that the username or password is incorrect. This feedback is important to help the user understand what went wrong during the login process.

At this stage, users should be able to log in successfully, and if any errors occur, they will receive relevant feedback in the form of a message.

5.2.3 Creating a Login Template

After creating the form and view to handle login, the next step is to create the HTML template to display the login form. This template will be used to accept input from users, such as their username and password. We will also add interactive elements and styling to ensure the login form looks appealing and is easy to use.

Creating an HTML Template for the Login Form

To begin, create a `login.html` file in the `templates/account/` folder. This template will render the login form that we previously created in the view.

Here is an example of the template for the login page:

```
<!-- File: apps/templates/account/login.html -->

{% extends 'home/base.html' %}
{% block title %}Login - Bot Platform{% endblock %}
{% block content %}
    <div class="container">
        <div class="login-wrapper">
            <h2 class="login-title">Login</h2>

            <!-- Displaying messages if there are any
errors or success -->
            {% if messages %}
                <div class="messages">
                    {% for message in messages %}
                        <div class="alert
{{ message.tags }}">{{ message }}</div>
                    {% endfor %}
                </div>
            {% endif %}

            <!-- Login Form -->
            <form method="POST">
                {% csrf_token %}

                <!-- Username field -->
                <div class="form-group">
                    {{ form.username.label_tag }}
                    {{ form.username }}
                </div>

                <!-- Password field -->
                <div class="form-group">
                    {{ form.password.label_tag }}
                    {{ form.password }}
                </div>

                <!-- Submit button -->
                <div class="form-group">
                    <button type="submit" class="btn
btn-primary">Login</button>
                </div>
            </form>

            <!-- Link for users who do not have an
account -->
            <div class="register-link">
```

```
                <p>Don't have an account? <a href="{%
url 'register' %}">Sign up now</a>.</p>
            </div>
        </div>
    </div>
{% endblock %}
```

Explanation of the Login Template

- ***Using CSS and Static Assets***: In the `<head>` section, we link the template with an external CSS file using `{{ ASSETS_ROOT }}`, which is defined in the login view to refer to the static folder. This ensures that the login form gets consistent styling with other parts of the application.
- ***Message Display***: We use the `if messages` block to display any error or success messages sent from the view. This helps provide real-time feedback to users when they try to log in.
- ***Login Form***: The template renders `form.username` and `form.password` from the `UserLoginForm` in the view. Each field is placed inside a `div` with the `form-group` class to ensure the form elements are properly organized.
- ***CSRF Token***: Don't forget to include `{% csrf_token %}` inside the form. This is a security feature provided by Django to prevent Cross-Site Request Forgery (CSRF) attacks.
- ***Submit Button***: The login button is created using the `<button>` element with the `btn btn-primary` class. This class indicates that we are using a CSS framework like Bootstrap, but you can customize it

according to the framework or manual styles you are using.
- ***Sign-Up Link***: If the user doesn't have an account, we add a link that will direct them to the registration page using `{% url 'register' %}`.

5.2.4 Adding Login URL Routing

After completing the setup of the form, view, and template for login, the next step is to add the URL routing for the login page.

Add the following path in the `urls.py` file of the `authentications` app:

```
# File: apps/authentications/urls.py

from django.urls import path
from .views import user_login

urlpatterns = [
    path('login/', user_login, name='login'),
]
```

Run your local server using the following command:

```
$ python manage.py runserver
```

Then, open your browser and access the login page. Try logging in with both correct and incorrect credentials to test the system's authentication validity and error handling.

5.2.5 Testing the Login Feature (Optional)

After completing the creation of the login form, login view, and its template, the next step is to test the login feature. This testing ensures that the login feature works as expected, including handling different login scenarios that might occur. We will use Django's test framework to systematically and efficiently test the login feature.

Testing Login with Various Scenarios

The first step in testing the login feature is to ensure that users can log in with valid information and receive an error message if they use invalid information. Several key scenarios to test include:

- ***Successful Login Scenario***: The user enters the correct username and password and successfully logs in.
- ***Failed Login Scenario***: The user enters the wrong username or password and receives an error message.
- ***Empty Input Scenario***: The user attempts to log in without filling out any fields and receives appropriate validation messages.

Setting Up Tests with Django Test Framework

Django provides a built-in test framework that is very useful for automatically testing applications. To begin, create a new file within the application's directory to store these tests.

For example, in the `authentications` app, create a file named `test_views.py` inside the `tests/` folder. Django usually generates the `tests/` directory automatically when the project is started, but if it does not exist, you can create it manually.

```
$ mkdir -p apps/authentications/tests
$ touch apps/authentications/tests/test_views.py
```

Writing Test Cases for the Login Feature

Once the test file is ready, we can start writing test cases for the login feature. These tests will use Django's `TestCase` to simulate HTTP requests and evaluate the responses from the login view.

Here's an example of tests for the login feature:

```
# File: apps/authentications/tests/test_views.py
from django.test import TestCase
from django.urls import reverse
from django.contrib.auth.models import User

class UserLoginTest(TestCase):

    def setUp(self):
        # Create a user to test login
        self.user =
User.objects.create_user(username='testuser',
password='password123')

    def test_login_page_renders_correctly(self):
        # Test if the login page is rendered
correctly
        response = self.client.get(reverse('login'))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response,
'account/login.html')

    def test_login_successful(self):
        # Test if the user can log in with correct
credentials
        response = self.client.post(reverse('login'),
{
```

```
            'username': 'testuser',
            'password': 'password123'
        })
        self.assertRedirects(response,
reverse('dashboard'))  # Test if the user is
redirected to the dashboard

    def test_login_invalid_credentials(self):
        # Test if the user fails to log in with wrong
credentials
        response = self.client.post(reverse('login'),
{
            'username': 'testuser',
            'password': 'wrongpassword'
        })
        self.assertEqual(response.status_code, 200)
# User stays on the login page
        self.assertContains(response, 'Invalid
username or password')  # Error message is displayed

    def test_login_empty_fields(self):
        # Test if the user fails to log in without
entering any data
        response = self.client.post(reverse('login'),
{
            'username': '',
            'password': ''
        })
        self.assertEqual(response.status_code, 200)
# User stays on the login page
        self.assertContains(response, 'This field is
required')  # Validation message is displayed
```

Test Explanation

- ***`setUp` Method***: This method is executed before each test case. Here, we create a user (`testuser`) for the purpose of testing the login feature.
- ***Login Page Test***: The first test (`test_login_page_renders_correctly`)

ensures that the login page is rendered correctly and uses the appropriate template (`login.html`).

- ***Successful Login Test***: The second test (`test_login_successful`) checks if a user can log in with the correct credentials and if they are redirected to the dashboard after a successful login.
- ***Failed Login Test (Invalid Credentials)***: The third test (`test_login_invalid_credentials`) ensures that users who enter the wrong username or password receive an error message and remain on the login page.
- ***Failed Login Test (Empty Fields)***: The last test (`test_login_empty_fields`) verifies that users who try to log in without filling in any fields will see the appropriate validation messages.

Running the Tests

Once we've written the test cases, we can run the tests using the following command in the terminal:

```
$ python manage.py test authentications
```

This command will run all tests in the `authentications` app, including the tests for the login feature we just created. If all tests pass, we'll see output indicating that all tests have succeeded.

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------
----------------
Ran 4 tests in 0.123s
```

```
OK
```

Handling Errors or Failures

If any test fails, Django will provide a description of the error, such as an incorrect HTTP status code or an error message not being displayed as expected. We can fix the issue in the application code or the test case, then rerun the tests until all tests pass.

Testing is crucial to ensure that the login feature functions correctly in various scenarios and conditions. By using Django's test framework, we can automate these tests, making the process more efficient and accurate in ensuring the application's functionality.

5.3 Creating Logout Feature

After the login feature is functioning well and has been tested, we will proceed to add a logout feature to our application. This feature is important for users to securely log out of their accounts and ensure their sessions are properly terminated. In Django, the logout process is quite simple because Django provides a logout function that can handle various aspects such as clearing the user's session.

5.3.1 Setting Up the Logout View

To start, we will create a view for logging out. This view will call Django's logout function, which automatically deletes the user's session and cookies, so the user is no longer considered logged in. After logging out, we will redirect the user back to the login page or another appropriate page.

The logout view is very straightforward. First, we will add a `user_logout` view that calls the logout function to end the user's session and then redirects them back to the login page. The code for the logout view that we have prepared is as follows:

```
# File: apps/authentications/views.py

from django.contrib.auth import logout
from django.shortcuts import redirect

# View for logging out users
def user_logout(request):
    # The logout function will remove the user's
session
    logout(request)
    # After logging out, the user is redirected back
to the login page
    return redirect('login')
```

- ***Function logout(request)***: This function will clear the user's session, including removing cookies related to the login. Users who have logged out will not have access to pages that require authentication.
- ***Redirection***: After the logout process is complete, we use `redirect('login')` to direct the user to the login page. You can also change this redirection page to another page, such as the home page or a logout information page that informs the user that they have successfully logged out.

5.3.2 Setting Up URL Routing for Logout

After the logout view has been created, we need to add a URL that will map to this view. Insert the logout URL into the `urls.py` file of the `authentications` application.

```
# File: apps/authentications/urls.py

from django.urls import path
from .views import user_logout

urlpatterns = [
    path('logout/', user_logout, name='logout'),
    # Other URLs...
]
```

By adding this path, users can access the logout page via the URL `/logout/`. The `user_logout` view will be executed each time that URL is accessed, and users will log out of their session.

Handling Sessions and Cookies (Optional)

When users log out, Django automatically handles the deletion of the session and cookies associated with authentication. A session is the mechanism used to store the user's login status, and when the user logs out, the session is deleted.

Additionally, if your application uses cookies to store additional information, such as user preferences, those cookies will not be deleted by the logout. Therefore, if you have custom cookies that you want to delete upon logout, you can add that manually in the `user_logout` view.

Example of deleting custom cookies:

```
# File: apps/authentications/views.py

from django.contrib.auth import logout
from django.shortcuts import redirect

def user_logout(request):
    # Log out the user
    logout(request)

    # Delete custom cookies (e.g., 'remember_me')
    if 'remember_me' in request.COOKIES:
        response = redirect('login')
        response.delete_cookie('remember_me')
        return response

    # After logging out, redirect the user back to
the login page
    return redirect('login')
```

In the example above, in addition to removing the session, we also delete the `remember_me` cookie if it exists. This ensures that all data related to the user's session is deleted when they log out.

Adding a Logout Button in the Template

To allow users to easily log out, we need to add a logout button in the template, for example in the header or navigation menu, so it is always visible to users who are logged in. This button will link to the URL `/logout/`.

Example HTML code to add a logout button:

```
<!-- File: templates/base.html -->

{% if user.is_authenticated %}
    <a href="{% url 'logout' %}" class="btn btn-
outline-danger">Logout</a>
{% endif %}
```

In the template above, we use `{% if user.is_authenticated %}` to check if the user is logged in. If the user is logged in, the logout button will be displayed. When the user clicks this button, they will be redirected to the URL `/logout/`, which executes the `user_logout` view.

5.3.3 Setting Redirect After Logout

After successfully implementing the logout feature, we can adjust the page that users are directed to after they log out. By default, in the previous example, users are redirected to the login page after logging out. However, depending on the needs of the application, we can set it so that users are redirected to another page, such as the home page or a home page.

Determining the target page after logout is quite flexible, and we can do it easily using the `redirect` function. If we want to redirect users to the home page after logging out, simply replace `redirect('login')` with `redirect('home')` or another appropriate page.

Here is a simple modification to the logout view to redirect users to the home page:

```
# File: apps/authentications/views.py
```

```
from django.contrib.auth import logout
from django.shortcuts import redirect

def user_logout(request):
    # Log out the user
    logout(request)
    # After logging out, redirect the user to the
home page
    return redirect('login')  # Change to 'home' or
another desired page
```

In this example, after users log out, they will be redirected to the login page (login). This page is specified through the login URL path, which must have been previously defined in your application's `urls.py`. If the login page does not exist, ensure to create it.

Determining the logout target page is very important to enhance the user experience. Redirecting them to a relevant page or presenting a message "You have successfully logged out" can provide a better interaction feel.

If your application uses a frequently accessed home or dashboard page, this setting could be a good choice to maintain user navigation flow after they log out of their account.

5.3.4 Testing the Logout Feature (Optional)

After implementing the logout feature, it is important to test it with various scenarios to ensure that the logout function works correctly and that the user session is properly closed. Django provides a testing framework that we can use to automate this testing process.

Using Django’s Test Framework

To test the logout feature, we will create a test case that verifies whether the user session is deleted and whether the user is redirected to the correct page after logging out. This testing will be done using Django's test framework in the `tests.py` file.

Steps to test logout:

- The user logs in first.
- The user accesses the logout URL.
- The system ensures the user session is deleted, and they are no longer considered an authenticated user.
- Ensure the user is redirected to the appropriate page (e.g., home or login).

Example test:

```
# File: apps/authentications/tests.py

from django.test import TestCase
from django.urls import reverse
from django.contrib.auth.models import User

class LogoutTestCase(TestCase):
    def setUp(self):
        # Create a new user for testing
        self.user =
User.objects.create_user(username='testuser',
password='password123')

    def test_logout(self):
        # Log in the user before logging out
        self.client.login(username='testuser',
password='password123')

        # Ensure the user is authenticated
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
self.assertTrue(response.context['user'].is_authentic
ated)

        # Access the logout URL
        response = self.client.get(reverse('logout'))

        # Ensure the user is redirected to the
correct page after logging out
        self.assertRedirects(response,
reverse('home'))

        # Ensure the user is no longer authenticated
        response = self.client.get(reverse('home'))

self.assertFalse(response.context['user'].is_authenti
cated)
```

- ***SetUp Method:*** At the beginning of the test, we create a new user (testuser) with a specific password. This user is used in each logout test.
- ***Login Test:*** We log in using `self.client.login()` and ensure that the user successfully logs in and is authenticated.
- ***Logout Test:*** After logging in, we access the logout URL using `self.client.get(reverse('logout'))` to perform the logout.
- ***Redirection and Session:*** The test ensures the user is redirected to the home page after logging out and checks whether the user session has been deleted, confirming that the user is no longer authenticated.

Using Terminal Commands

Once the logout test code is ready, run the tests using terminal commands to ensure all scenarios work correctly. Run the following command from your project directory:

```
$ python manage.py test
```

This command will run all tests present in the `tests.py` file, including the logout feature tests. If all tests pass, the terminal output will indicate that all features are functioning correctly, including logout.

Additional Testing Scenarios

In addition to the basic scenarios above, you can also test logout with various variations, such as:

- Logging out without logging in first: Accessing the logout URL without logging in and ensuring that the system does not encounter errors and that the user is still redirected to the correct page.
- Ensuring there is no access after logout: Accessing pages that require authentication after logging out to ensure that the user can no longer access them.

By performing these tests, you can ensure that the logout feature works perfectly, deletes the user session, and redirects them to the appropriate page after logging out. This is essential for ensuring the security of the application, especially when handling sensitive user sessions.

5.4 Complete Code

Before proceeding, we will review the code we have created in our authentication app to ensure there are no errors in the code writing. Below is the complete code.

5.4.1 Models.py

```
# File: apps/authentication/models.py

from django.contrib.auth.models import User
from django.db import models
from django.utils import timezone

class UserProfile(models.Model):
    user = models.OneToOneField(User,
on_delete=models.CASCADE)
    city = models.CharField(max_length=100,
blank=True)
    address = models.CharField(max_length=255,
blank=True)
    country = models.CharField(max_length=100,
blank=True)
    postal_code = models.CharField(max_length=20,
blank=True)
    about = models.TextField(blank=True)
    company = models.CharField(max_length=100,
blank=True)
    profile_picture =
models.ImageField(upload_to='profile_pics/',
blank=True)

    def __str__(self):
        return self.user.username
```

5.4.2 Forms.py

```
# File: apps/authentication/forms.py
```

```
from django import forms
from django.contrib.auth.forms import
UserCreationForm, AuthenticationForm
from django.contrib.auth.models import User

class UserRegistrationForm(UserCreationForm):
    email =
forms.EmailField(widget=forms.EmailInput(attrs={'clas
s': 'form-control', 'placeholder': 'Email'}))
    username =
forms.CharField(widget=forms.TextInput(attrs={'class'
: 'form-control', 'placeholder': 'Username'}))
    password1 =
forms.CharField(widget=forms.PasswordInput(attrs={'cl
ass': 'form-control', 'placeholder': 'Password'}))
    password2 =
forms.CharField(widget=forms.PasswordInput(attrs={'cl
ass': 'form-control', 'placeholder': 'Confirm
Password'}))
    first_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'First Name'}))
    last_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Last Name'}))
    company = forms.CharField(max_length=100,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Company'}))
    address = forms.CharField(max_length=255,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Address'}))
    city = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'City'}))
    country = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Country'}))
    postal_code = forms.CharField(max_length=20,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Postal Code'}))
```

```
    about =
forms.CharField(widget=forms.Textarea(attrs={'class':
'form-control', 'placeholder': 'About me'}),
required=False)
    profile_picture =
forms.ImageField(required=False,
widget=forms.FileInput(attrs={'class': 'form-control-
file'}))

    class Meta:
        model = User
        fields = ['username', 'email', 'password1',
'password2', 'first_name', 'last_name', 'company',
'address', 'city', 'country', 'postal_code', 'about',
'profile_picture']


# Form for user login
class UserLoginForm(AuthenticationForm):
    # Field for username
    username = forms.CharField(
        widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Username'})
    )
    # Field for password
    password = forms.CharField(
        widget=forms.PasswordInput(attrs={'class':
'form-control', 'placeholder': 'Password'})
    )
```

5.4.3 Views.py

```
# File: apps/authentication/views.py

from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login,
logout
from django.contrib import messages
from .forms import UserRegistrationForm,
UserLoginForm
from .models import UserProfile
```

```
# View for the home page
def home(request):
    context = {
        'ASSETS_ROOT': '/static/assets',  #
Additional path for static assets if needed
    }
    return render(request, 'home/home.html', context)


# View for handling new user registration
def user_register(request):
    if request.method == 'POST':
        # Check if the request is POST and process
the form data
        form = UserRegistrationForm(request.POST,
request.FILES)
        if form.is_valid():
            # Save user data to the User model
            user = form.save()
            # Create and save user profile linked to
User
            UserProfile.objects.create(
                user=user,
                city=form.cleaned_data.get('city',
''),

address=form.cleaned_data.get('address', ''),

country=form.cleaned_data.get('country', ''),

postal_code=form.cleaned_data.get('postal_code', ''),
                about=form.cleaned_data.get('about',
''),

profile_picture=request.FILES.get('profile_picture',
None)
            )
            # Automatically log in the user after
registration
            login(request, user)
            # Redirect to home or dashboard page
after successful registration
```

```
            return redirect('home')
    else:
        # If the request is not POST, create an empty
form to display on the page
        form = UserRegistrationForm()

    # Prepare context for the template
    context = {
        'form': form,
        'ASSETS_ROOT': '/static/assets',  # Including
path for assets
    }

    # Render registration page with the prepared form
    return render(request, 'account/register.html',
context)


# View for user login
def user_login(request):
    if request.method == 'POST':
        # Retrieve data from the form and validate
        form = UserLoginForm(data=request.POST)
        if form.is_valid():
            # Authenticate user
            user =
authenticate(username=form.cleaned_data['username'],
password=form.cleaned_data['password'])
            if user is not None:
                # Log in user if data is valid
                login(request, user)
                messages.success(request, 'Login
successful.')
                return redirect('home')  # Redirect
user after successful login
            else:
                # Display error message if
authentication fails
                messages.error(request, 'Invalid
username or password.')
    else:
        form = UserLoginForm()
    context = {
        'ASSETS_ROOT': '/static/assets',  # 
Additional path for static assets if needed
        'form': form,
    }
    return render(request, 'account/login.html', 
context)

# View for user logout
def user_logout(request):
    # Log out user
    logout(request)

    # Delete custom cookies (e.g., 'remember_me')
    if 'remember_me' in request.COOKIES:
        response = redirect('login')
        response.delete_cookie('remember_me')
        return response

    # After logout, redirect user back to the login 
page
    return redirect('login')
```

5.4.4 URLs.py

```
# File: apps/authentication/urls.py

from django.urls import path
from .views import home, user_register, user_login, 
user_logout

urlpatterns = [
    path('', home, name='home'),
    path('register/', user_register, 
name='register'),
    path('login/', user_login, name='login'),
    path('logout/', user_logout, name='logout'),
]
```

With all this code, you have a comprehensive implementation of the authentication features in your Django application, covering user registration, login, logout, and user profile management. This complete setup ensures that your application can handle user sessions securely and efficiently.

5.5 Connecting The Home Page With The Authentication App

First, we need to ensure that the URLs leading to the **Login** and **Registration** pages are properly defined in the `urls.py` file of the Authentication app. Let’s review how to do this.

Linking URLs from the Home Page

On the **Home** page, we already have a button labeled "**Start Now**" that will direct users to the **Registration** page. We can also add a link to the **Login** page for users who already have an account.

To do this, we need to ensure that the URLs for **Login** and **Registration** are correctly defined in the `urls.py` file of the **Authentication** app.

In the template file for the **Home** page located at **`apps/templates/authentication/home.html`**, we will update or add links to the **Login** and **Registration** pages. Add the links as follows:

```
<!-- File: apps/templates/authentication/home.html --
>

{% extends 'home/base.html' %}

{% block title %}Home - Bot Platform{% endblock %}

{% block content %}
<div class="jumbotron text-center">
    <h1 class="display-4">Welcome to the Bot
Platform</h1>
    <p class="lead">Your gateway to managing and
interacting with your bots.</p>
    <hr class="my-4">
    <p>Create and manage your bots easily through our
platform.</p>
    <a class="btn btn-primary btn-lg" href="{% url
'register' %}" role="button">Start Now</a>
    <p class="mt-4">Already have an account? <a
href="{% url 'login' %}">Log in here</a></p> <!--
Link to the Login page -->
</div>

<div class="row">
    <div class="col-md-4">
        <h3>Easy to Use</h3>
        <p>Our platform is designed to be simple and
intuitive, allowing you to create and manage bots
effortlessly.</p>
    </div>
    <div class="col-md-4">
        <h3>Flexible</h3>
        <p>Customize your bots to meet your needs
with our flexible settings and tools.</p>
    </div>
    <div class="col-md-4">
        <h3>Scalable</h3>
        <p>Whether you are managing one bot or
hundreds, our platform can scale to meet your
business needs.</p>
    </div>
</div>
{% endblock %}
```

In this template, we added the link `href="{% url 'login' %}"` that directs users to the **Login** page if they already have an account. This link uses Django's URL tag to refer to the URL name we registered in `urls.py`.

Finally, if users click "**Start Now**," they will be directed to the **Registration** page, and if they click "**Log in here**," they will be directed to the **Login** page.

In the **`base.html`** file, we will also add links for **registration**, **login**, and **logout** in the navbar.

```
<!-- File: apps/templates/home/base.html -->

<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Bot Platform{% endblock
%}</title>
    <!-- Adding Bootstrap from CDN -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhj
Y6hW+ALEwIH" crossorigin="anonymous">
</head>
<body>
    <!-- Header section for navigation -->
    <header>
        <nav class="navbar navbar-expand-lg navbar-
light bg-light">
            <a class="navbar-brand" href="{% url
'home' %}">Bot Platform</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 ms-auto">
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'login' %}">Log In</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'register' %}">Sign Up</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'logout' %}">Log Out</a>
                    </li>
                </ul>
            </div>
        </nav>
    </header>
    <main class="container mt-5">
        {% block content %}
        {% endblock %}
    </main>
    <footer class="text-center py-4 bg-dark text-
white">
        <p>&copy; 2024 Bot Platform. All rights
reserved.</p>
    </footer>
    <!-- Including Bootstrap JavaScript -->
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11
.8/dist/umd/popper.min.js" integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc
2pM8ODewa9r" crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/js/bootstrap.min.js" integrity="sha384-
```

```
0pUGZvbkm6XF6gxjEnlmuGrJXVbNuzT9qBBavbLwCsOGabYfZo0T0
to5eqruptLy" crossorigin="anonymous"></script>
</body>
</html>
```

With these steps, we have connected the **Home** page to the **Login** and **Registration** pages in the **Authentication** app. Users can now easily navigate from the main page to the appropriate page based on their status, whether they want to create a new account or log in to an existing one.

Chapter Conclusion

In this chapter, we have thoroughly discussed the process of creating user authentication features, including registration, login, and logout, in the platform_bot project. Starting from the creation of registration forms and templates that allow users to create accounts, we also implemented validation and testing to ensure that the features work as expected.

Next, the login feature was developed to allow users to access their accounts with strong authentication. This login process is complemented by strict validation and testing to ensure smooth login, whether the data is correct or when input errors occur.

The logout feature was then implemented to give users the ability to securely exit their accounts. We also added a redirect setting after logout to enhance user experience, ensuring they are directed to relevant pages. Furthermore, testing of the logout feature was conducted to ensure that user sessions are correctly terminated after logout.

In **Sub-section 5.1**, we began by creating a user model that allows us to store and manage user data effectively. This sub-section includes an explanation of how to define the user model, add relevant fields, and tailor the model to the application's needs.

Sub-section 5.2 focuses on the creation of the login feature. By discussing the creation of login forms, appropriate templates, and

URL configuration, we can understand how the user authentication process is carried out and how to ensure a user-friendly login experience.

In **Sub-section 5.3**, we discussed the important logout feature to ensure users can safely exit the system. The steps described include setting up views for logout, configuring URLs, and handling redirects after logout to enhance user experience.

Overall, this chapter provides a comprehensive guide to implementing basic authentication features in Django applications and integrating them well within a broader system. By understanding and applying these techniques, you can build applications that are more secure and user-friendly, as well as ready for integration with additional features that may be needed in the future.

Each step has been accompanied by testing practices using Django's test framework, ensuring that the entire authentication process runs smoothly without issues. With this complete authentication feature, the platform_bot application has a strong foundation for handling user security and safety.

Chapter 6 - Creating the User Dashboard Page

In the development of the platform_bot application, the dashboard plays a crucial role as the control center for registered users. The dashboard allows users to manage their accounts, access bot features, and monitor performance and statistics. In this section, we will discuss how to design a user-friendly dashboard layout that simplifies navigation and accessibility for users.

6.1 Understanding The Purpose And Benefits Of The Dashboard

The dashboard is the main page users will access after successfully logging in. Its primary goal is to provide an overview of important data or activities relevant to the user. In the context of building bots using Django, the dashboard can be used to display information such as bot usage statistics, running bot status, recent activity logs, and a menu for managing the bot.

The benefits of a dashboard include:

1. ***Centralized Management:*** The dashboard enables users to manage all aspects of their bots in one place without having to switch between different pages.

2. ***Quick Access to Information:*** Users can quickly access important information about their bots, such as bot status, usage statistics, and more.
3. ***Easy Navigation:*** With a sidebar or navigation menu, users can easily switch between different features available in the application.
4. ***Improved User Experience:*** A well-designed dashboard can enhance the user experience, making the application feel more professional and easy to use.

We will begin with practical steps to create this dashboard page, starting from the model (if needed), view, template, routing, and styling. All these components will come together to form a functional and attractive dashboard page.

Next, we will discuss creating additional models for the information to be displayed on the dashboard, if needed.

6.2 Creating Additional Models For Dashboard Information

At this stage, we will create additional models to store information that will be displayed on the dashboard. In many cases, the dashboard will display data that already exists in the application, such as user profiles, usage activity, etc. However, if there is specific information that needs to be displayed on the dashboard and is not covered by existing models, we may need to add a new model.

Determining the Need for Additional Models

First, we need to determine what information will be displayed on the dashboard. If that information is already covered by models we've previously created, there's no need to create an additional model. However, if there's special data not covered, such as daily statistics, activity logs, or notifications, we need to create a new model.

Example case:

- We want to display an activity log performed by users on the dashboard. We need to create a new model to store this log.

Creating an Activity Log Model

To store user activity logs, we can create a new model like this:

```
# File: apps/dashboard/models.py
from django.contrib.auth.models import User
from django.db import models
from django.utils import timezone

class ActivityLog(models.Model):
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # Relationship to the User
model
    action = models.CharField(max_length=255)  #
Description of the activity
    timestamp =
models.DateTimeField(default=timezone.now)  # Time
when the activity occurred

    def __str__(self):
        return f"{self.user.username} - {self.action}
at {self.timestamp}"
```

The comments above explain each element of the model:

- ***user:*** Stores a reference to the user who performed the activity.
- ***action:*** Stores a short description of the activity performed.
- ***timestamp:*** Stores the time when the activity occurred.

This model can be used to record all user activities within the application, which can then be displayed on the dashboard.

Applying Migrations for the New Model

After we add the new model in `users/models.py`, we need to create and apply migrations to update the database schema.

In the terminal, run the following command to create the migration:

```
$ python manage.py makemigrations
```

This command will create a migration file for the `ActivityLog` model. Next, apply the migration to the database with the following command:

```
$ python manage.py migrate
```

Once the migration is complete, the database schema will be updated, and we are ready to use the `ActivityLog` model to record user activities.

Integrating the Model with the Dashboard

The `ActivityLog` model we just created can be integrated into the dashboard to display recent user activities. In the following section, we will create views and templates to display this information on the dashboard page.

By following these steps, we have successfully created the additional model needed to display information on the dashboard. Now, we proceed to create the views and templates for the dashboard.

6.3 Creating The Dashboard View

After designing a dashboard layout that includes navigation and accessibility, the next step is to display the relevant data within it. The view for the dashboard is responsible for fetching and displaying appropriate data for the user. Additionally, we must ensure that only users with proper access rights can view and manage the information on the dashboard.

The view is a crucial component in Django's application architecture, acting as a bridge between the business logic and the presentation (template). In the context of the user dashboard, the view is responsible for fetching data related to the logged-in user, such as profile information, activity logs, and bot statistics, then rendering a template to display it.

Here’s how to create a view for the dashboard that displays bot data and manages user access rights:

```
# File: apps/dashboard/views.py
```

```
from django.shortcuts import render
from django.contrib.auth.decorators import
login_required
from apps.authentication.models import UserProfile
from apps.dashboard.models import ActivityLog

@login_required
def dashboard(request):
    # Fetch the profile of the currently logged-in
user
    user_profile =
UserProfile.objects.filter(user=request.user).first()

    # Fetch the activity log of the currently logged-
in user
    activity_logs =
ActivityLog.objects.filter(user=request.user).order_b
y('-timestamp')

    # Data to pass to the template
    context = {
        'ASSETS_ROOT': '/static/assets',
        'user_profile': user_profile,
        'activity_logs': activity_logs,
    }

    # Render the dashboard.html template with the
fetched data
    return render(request,
'dashboard/dashboard.html', context)
```

In the `dashboard` view, we use the `@login_required` decorator to ensure that only logged-in users can access the dashboard page. If a user who is not logged in attempts to access this page, they will be redirected to the login page.

Next, we retrieve data from the `UserProfile` and `ActivityLog` models. `UserProfile` contains personal information about the user, which will be displayed on the dashboard, such as their name, profile picture, and other details. Meanwhile, `ActivityLog` stores records of the user's activities, like interactions with the bots they have created. We use the `filter()` method on the `ActivityLog` model to retrieve all activities associated with the currently logged-in user, ordered by the most recent `timestamp`.

Access Control

Access control is managed using the `@login_required` decorator, which ensures that only logged-in users can view the dashboard page. This is important for maintaining the privacy and security of user data. By implementing this login guard, the application will not allow access without valid authentication.

After fetching the data, we pass it to the template by creating a context that includes `user_profile` and `activity_logs`. The `dashboard.html` template will then be responsible for displaying this data in a way that is easy for the user to understand.

Access Testing

We can test whether this feature works correctly by trying to access the dashboard URL without being logged in. If the user is redirected to the login page, it means access control has been properly implemented.

6.4 Creating The Dashboard Page Template

The next step in building the dashboard is to determine the main elements that will be displayed. In the platform_bot application.

some features that will be available on the dashboard include:

- ***Activity Log:*** Displays information about user activity logs, such as when a user edits their profile.
- ***Bot Navigation:*** A menu to manage bots, such as creating new bots, editing existing bots, or monitoring running bots.
- ***Settings:*** Access to account settings and user preferences.
- ***Logout:*** A button to log out of the system.

Once these elements are determined, the next step is to design a layout that is clear and easy to navigate. The dashboard design should be intuitive, ensuring that users can quickly understand the functionality of each feature.

For example, the dashboard header can display the user's name along with a dropdown button for settings and logout. On the left side, there can be a vertical navigation menu for main features like the **Main Dashboard**, **Bot Management**, and **Statistics**. Meanwhile, the right side (content area) will be used to display specific content for the selected feature.

6.4.1 Creating the `base.html` Template

First, we will create the `base.html` template, which will serve as the foundation for all the pages in the dashboard. This template will structure the basic HTML, including the header, footer, and main content area.

Create a `base.html` file in the `apps/templates/dashboard/` directory with the following content:

```
<!-- File: apps/templates/dashboard/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>Dashboard | Platform Bot</title>
    <!-- Adding Bootstrap from CDN -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhj
Y6hW+ALEwIH" crossorigin="anonymous">
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.10.5/font/bootstrap-icons.css"
rel="stylesheet">
    <link
href="https://stackpath.bootstrapcdn.com/bootstrap/5.
3.0/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-light
bg-light">
        <a class="navbar-brand" href="{% url 'home'
%}">Platform Bot</a>
```

```
        <div class="collapse navbar-collapse"
id="navbarNav">
            <ul class="navbar-nav ms-auto">
                <li class="nav-item">
                    <a class="nav-link" href="{% url
'logout' %}">Logout</a>
                </li>
            </ul>
        </div>
    </nav>
    <div class="container-fluid">
        <div class="row">
            <div class="col-md-3">
                <!-- Sidebar -->
                <div class="list-group">
                    <a href="{% url 'dashboard' %}"
class="list-group-item list-group-item-
action">Dashboard</a>
                    <a href="#" class="list-group-
item list-group-item-action">Profile</a>
                    <a href="#" class="list-group-
item list-group-item-action">Create Bot</a>
                    <a href="#" class="list-group-
item list-group-item-action">Manage Bot</a>
                </div>
            </div>
            <div class="col-md-9">
                <!-- Main Content -->
                {% block content %}
                {% endblock %}
            </div>
        </div>
    </div>

    <!-- Including Bootstrap JavaScript -->
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11
.8/dist/umd/popper.min.js" integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc
2pM8ODewa9r" crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/js/bootstrap.min.js" integrity="sha384-
```

```
0pUGZvbkm6XF6gxjEnlmuGrJXVbNuzT9qBBavbLwCsOGabYfZo0T0
to5eqruptLy" crossorigin="anonymous"></script>

</body>
</html>
```

This design uses a standard structure that separates the header, a sidebar for navigation, and the main content area. This template can be used as the base for all pages within the dashboard, leveraging Django Template Inheritance to maintain design consistency across pages.

Additionally, it is important to ensure that the dashboard layout is responsive, so it can be accessed properly on both mobile and desktop devices.

6.4.2 Creating the `dashboard.html` Page

Next, we will create the `dashboard.html` template that will use `base.html` as the base layout and display the data we prepared in the dashboard view.

We will also add code to display success and error messages when the user logs in.

```
<!-- Displaying messages for error or success -->
{% if messages %}
    <div class="messages">
        {% for message in messages %}
            <div class="alert {{ message.tags
}}">{{ message }}</div>
        {% endfor %}
    </div>
{% endif %}
```

With this code, the dashboard will display messages for the user if the login was successful.

Create the `dashboard.html` file in the directory `apps/templates/dashboard/` with the following content:

```
<!-- File: apps/templates/dashboard/dashboard.html --
>
{% extends 'dashboard/base.html' %}

{% block content %}
<h1 class="fw=bold">Dashboard</h1>

<h2>Welcome to Your Dashboard,
{{ user_profile.user.username }}</h2>
<p>On this Dashboard, we have already displayed the
activity log and user profile. Next, we will add more
features.</p>

<!-- Displaying messages for error or success -->
{% if messages %}
    <div class="messages">
        {% for message in messages %}
            <div class="alert {{ message.tags
}}">{{ message }}</div>
        {% endfor %}
    </div>
{% endif %}

<h3>Activity Log</h3>
<table class="table table-bordered">
    <thead>
        <tr>
            <th>Timestamp</th>
            <th>Action</th>
        </tr>
    </thead>
    <tbody>
        {% for log in activity_logs %}
        <tr>
```

```
            <td>{{ log.timestamp }}</td>
            <td>{{ log.action }}</td>
        </tr>
        {% empty %}
        <tr>
            <td colspan="2">No recorded
activities.</td>
        </tr>
        {% endfor %}
    </tbody>
</table>

<h2>User Profile</h2>
<p>Username: {{ user_profile.user.username }}</p>
<p>City: {{ user_profile.city }}</p>
<p>Address: {{ user_profile.address }}</p>
<p>Country: {{ user_profile.country }}</p>
<p>Postal Code: {{ user_profile.postal_code }}</p>
<p>About: {{ user_profile.about }}</p>
<p>Company: {{ user_profile.company }}</p>

{% if user_profile.foto_profile %}
    <img src="{{ user_profile.foto_profile.url }}"
alt="Profile Photo">
{% else %}
    <p>No profile photo.</p>
{% endif %}

{% endblock %}
```

Code Explanation:

- This template extends `base.html`, so it automatically inherits the layout from `base.html`.
- The `{% block content %}` section is filled with specific content for the dashboard page, including a welcome message showing the username and a table displaying the user’s activity log.

6.4.3 Creating a Sidebar with Icons and Responsive Design Using Bootstrap

Next, we will modify the sidebar on the dashboard page to be more appealing by adding icons from Bootstrap Icons. Additionally, we will ensure the sidebar is responsive for mobile devices. This sidebar will also have a toggle button for small devices, allowing users to show or hide the sidebar as needed.

HTML Structure for the Sidebar with Icons

To add icons to the sidebar, we will use Bootstrap Icons. The icons will be placed inside each link element in the sidebar, followed by the navigation text. With this approach, each sidebar item will have an icon relevant to its function.

```
<nav id="sidebar" class="col-md-3 col-lg-2 d-md-block
bg-light sidebar">
    <div class="position-sticky">
        <ul class="nav flex-column">
            <li class="nav-item">
                <a class="nav-link active" href="{%
url 'dashboard' %}">
                    <i class="bi bi-house-door"></i>
Dashboard
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="{% url
'profile' %}">
                    <i class="bi bi-person"></i>
Profile
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">
                    <i class="bi bi-plus"></i> Create
Bot
```

```
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">
                    <i class="bi bi-gear"></i> Manage
Bots
                </a>
            </li>
        </ul>
    </div>
</nav>
```

In the above code, icons from Bootstrap Icons are added using the `<i>` element with the `bi` class. For instance, a house icon for the dashboard (`bi-house-door`), a person icon for the profile (`bi-person`), and so on.

Adding a Sidebar Toggle for Responsiveness

This sidebar must function well across different screen sizes, including mobile devices. For this, we will add a toggle button that appears only on small devices (screens below `md` or medium). This button will allow users to show or hide the sidebar more easily.

Here is the addition of the toggle button:

```
<div class="col-md-3">
    <button class="btn btn-primary d-md-none"
type="button" data-bs-toggle="collapse" data-bs-
target="#sidebarMenu" aria-expanded="false" aria-
controls="sidebarMenu">
        Menu
    </button>
    <div class="collapse d-md-block"
id="sidebarMenu">
        <div class="list-group">
```

```
            <a href="{% url 'dashboard' %}"
class="list-group-item list-group-item-action">
                <i class="bi bi-house-door"></i>
Dashboard
            </a>
            <a href="#" class="list-group-item list-
group-item-action">
                <i class="bi bi-person"></i> Profile
            </a>
            <a href="#" class="list-group-item list-
group-item-action">
                <i class="bi bi-plus"></i> Create Bot
            </a>
            <a href="#" class="list-group-item list-
group-item-action">
                <i class="bi bi-gear"></i> Manage
Bots
            </a>
        </div>
    </div>
</div>
```

In this code, the `<button>` element with the `d-md-none` class is used to display the button only on smaller screens. This button controls the collapse of the sidebar, targeting `#sidebarMenu`. When pressed, the sidebar will appear or disappear based on its collapse state.

Enhancing `base.html` for Full Responsiveness

Now, we will combine the two parts into a complete code to create a responsive sidebar with icons.

```
<!-- File: apps/templates/dashboard/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>Dashboard | Bot Platform</title>
    <!-- Adding Bootstrap and Bootstrap Icons from
CDN -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet">
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.10.5/font/bootstrap-icons.css"
rel="stylesheet">
</head>
<body>
    <!-- Navbar -->
    <nav class="navbar navbar-expand-lg navbar-light
bg-light">
        <div class="container-fluid">
            <a class="navbar-brand" href="{% url
'home' %}">Bot Platform</a>
            <button class="navbar-toggler"
type="button" data-bs-toggle="collapse" data-bs-
target="#sidebarMenu" aria-controls="sidebarMenu"
aria-expanded="false" aria-label="Toggle sidebar">
                <span class="navbar-toggler-
icon"></span>
            </button>
        </div>
    </nav>

    <div class="container-fluid">
        <div class="row">
            <!-- Sidebar -->
            <div class="col-md-3">
                <div class="collapse d-md-block"
id="sidebarMenu">
                    <nav id="sidebar" class="bg-light
sidebar">
                        <div class="position-sticky">
                            <ul class="nav flex-
column">
                                <li class="nav-item">
```

```
                                    <a class="nav-
link active" href="{% url 'dashboard' %}">
                                        <i class="bi
bi-house-door"></i> Dashboard
                                    </a>
                                </li>
                                <li class="nav-item">
                                    <a class="nav-
link" href="">
                                        <i class="bi
bi-person"></i> Profile
                                    </a>
                                </li>
                                <li class="nav-item">
                                    <a class="nav-
link" href="#">
                                        <i class="bi
bi-plus"></i> Create Bot
                                    </a>
                                </li>
                                <li class="nav-item">
                                    <a class="nav-
link" href="#">
                                        <i class="bi
bi-gear"></i> Manage Bots
                                    </a>
                                </li>
                                <!-- Logout button in
Sidebar -->
                                <li class="nav-item">
                                    <a class="nav-
link text-danger" href="{% url 'logout' %}">
                                        <i class="bi
bi-box-arrow-right"></i> Logout
                                    </a>
                                </li>
                            </ul>
                        </div>
                    </nav>
                </div>
            </div>

            <!-- Main Content -->
            <div class="col-md-9">
                {% block content %}
                {% endblock %}
            </div>
        </div>
    </div>

    <!-- Including Bootstrap JavaScript -->
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

This completes the creation of a sidebar with icons and a responsive design for your Django dashboard, making it functional on both desktop and mobile devices.

6.5 Setting Up URL Routing For The Dashboard

Now we need to set up URL routing to access the dashboard page. We will add the URL pattern in the `urls.py` file in the dashboard application.

Open or create the file `dashboard/urls.py` and add the following code:

```
# File: apps/dashboard/urls.py
from django.urls import path
from .views import dashboard

urlpatterns = [
    path('dashboard/', dashboard, name='dashboard'),
]
```

Code explanation:

- The `dashboard/` URL pattern maps to the `dashboard` view, so when users access `http://localhost:8000/dashboard/`, they will be directed to the dashboard page.

By adding this path, users can access the dashboard page via the `/dashboard/` URL. The `dashboard` view will run whenever this URL is accessed, and users will be logged out of their session.

Next, we need to ensure that the dashboard app's URL is included in the project's main routing. Open the `platform_bot/urls.py` file and add the following configuration:

```
# File: platform_bot/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('apps.dashboard.urls')),  # 
Including the dashboard app URL
]
```

With this configuration, the dashboard page will be displayed when users visit the root URL of the app.

6.6 Setting Up Automatic Redirect To Dashboard After Login

To solve the issue where users try to access pages that require authentication (like the dashboard page) without being logged in, you can redirect them to the login page or another page more effectively, rather than displaying a 404 error. You can use Django's mechanism to automatically redirect unauthenticated users to the correct login page.

Here’s how to fix it:

Set the Login URL in `settings.py`

Django has a special setting that can redirect users who aren’t logged in to the login page. You can make sure your login URL is correctly set in the `settings.py` file.

Open `settings.py` and add or modify the following line:

```
# settings.py

LOGIN_URL = '/login/'
```

By adding `LOGIN_URL`, Django will redirect unauthenticated users to the correct login page when they try to access pages that require login.

Redirect After Login

After a successful login, users are usually redirected back to the page they were trying to access before. Django automatically handles this through the `next` parameter. For example, if a user tries to access `/dashboard/`, they will be redirected to `/login/?next=/dashboard/`, and after logging in successfully, they will return to the dashboard.

If you want to configure a redirect page after login, you can use the following setting in `settings.py`:

```
# settings.py

LOGIN_REDIRECT_URL = '/dashboard/'  # The page to
redirect to after a successful login
```

With this setting, whenever a user logs in successfully, they will be redirected to the dashboard or another page you've specified.

These steps will ensure that unauthenticated users are redirected to the correct login page when trying to access pages that require authentication. By using `login_required` and properly setting the login URL, user experience is improved, and errors like 404 can be avoided.

Update Views in `apps/authentication`

We also need to update the views so that when a user registers or logs in, they will be redirected to the dashboard page.

Open the `views.py` file in `apps/authentication` and change the return redirect code in the `user_register` and `user_login` functions to point to the dashboard.

```
# File: apps/authentication/views.py
# View to handle new user registration
def user_register(request):

# ...

            # Redirect to the dashboard page after
successful registration
            return redirect('dashboard')
# ...

# View for user login
def user_login(request):
# ...
                return redirect('dashboard')  #
Redirect users after successful login

# ...
```

These steps will ensure that users who have registered or logged in will be redirected to the dashboard page.

Chapter Conclusion

This chapter has thoroughly explained the process of creating a user dashboard in a Django application. The main focus of this chapter is to provide a comprehensive guide on how to develop and manage an effective and informative dashboard for users.

Subsection 6.1 begins by outlining the purpose and benefits of the user dashboard, which helps establish the framework and necessary functionality. Understanding user needs and the goals of the dashboard is a crucial step to ensure it is both effective and valuable.

In **Subsection 6.1.2**, we discussed creating an additional model to store relevant information for the dashboard. This model enables us to store the data needed to display key information to users in real-time.

Subsection 6.1.3 explains the creation of the dashboard view, a critical component that connects the data model to the frontend display. This view is responsible for processing the data and providing it to the dashboard template.

In **Subsection 6.1.4**, we covered the creation of a template for the dashboard page. This template is designed to present data in an intuitive and engaging way, ensuring a smooth and interactive user experience.

Subsection 6.1.5 details the setup of URL routing for the dashboard, which is essential for easy access to the dashboard page through the appropriate URL. This setup ensures the dashboard can be accessed quickly and without difficulty.

Finally, **Subsection 6.1.6** addresses the automatic redirect to the dashboard after login. This feature enhances the user experience by directing users straight to their dashboard after successfully logging in, making navigation smoother.

Overall, this chapter provides a complete guide for creating and managing a user dashboard in a Django application, focusing on how to present data effectively and improve user interaction. By applying these techniques, you can build a dashboard that is not only functional but also adds significant value to your application's users.

Chapter 7 - User Profile Management

In this chapter, we will create user profile management functionality that allows users to view and edit their profile information. This page will include details such as username, email, first name, last name, address, city, country, postal code, company, about me, and profile picture.

7.1 Creating The User Profile Page

The `UserProfile` model we created in the previous chapter is sufficient to store user profile information. This model stores important data such as city, address, country, postal code, about the user, company, and profile picture.

In this section, we will create a page to display the user information stored in the `UserProfile` model.

7.1.1 Creating a View for the User Profile Page

First, we will set up a view to handle the logic for the user profile page. This view will handle retrieving the user's profile data and displaying the profile page.

Open the file `apps/users/views.py` and add the following code:

```
# File: apps/users/views.py

from django.shortcuts import render,
get_object_or_404
from django.contrib.auth.decorators import
login_required
from django.contrib.auth.models import User
from apps.authentication.models import UserProfile

@login_required
def profile(request):
    # Get the currently logged-in user
    user = request.user

    # Get the UserProfile associated with the user
    user_profile = get_object_or_404(UserProfile,
user=user)

    # Send user and user_profile data to the template
    context = {
        'user': user,
        'user_profile': user_profile,
    }

    return render(request, 'users/profile.html',
context)
```

In the code above, the `profile` function manages the display of the user profile page.

7.1.2 Creating the User Profile Page Template

Now, we need to create an HTML template for the user profile page.

The template for the profile page should include a form that displays user information such as username, email, first name, last name, and other profile details like address, company, and profile picture. Below is an example template for the user profile page:

Create a file named `profile.html` in the directory `apps/templates/dashboard/` and add the following code:

```
<!-- apps/templates/users/profile.html -->

{% extends "dashboard/base.html" %}

{% block content %}
<div class="container">
    <h2>User Profile</h2>
    <div class="row">
        <div class="col-md-8">
            <p><strong>Username:</strong>
{{ user.username }}</p>
            <p><strong>Email:</strong> {{ user.email
}}</p>
            <p><strong>First Name:</strong>
{{ user.first_name }}</p>
            <p><strong>Last Name:</strong>
{{ user.last_name }}</p>
            <p><strong>Address:</strong>
{{ user_profile.address }}</p>
            <p><strong>City:</strong>
{{ user_profile.city }}</p>
            <p><strong>Country:</strong>
{{ user_profile.country }}</p>
            <p><strong>Postal Code:</strong>
{{ user_profile.postal_code }}</p>
            <p><strong>Company:</strong>
{{ user_profile.company }}</p>
            <p><strong>About:</strong>
{{ user_profile.about }}</p>
        </div>
        <div class="col-md-4">
            {% if user_profile.foto_profile %}
```

```
                <img
src="{{ user_profile.foto_profile.url }}"
alt="Profile Picture" class="img-thumbnail">
            {% else %}
                <img src="" alt="Profile Picture"
class="img-thumbnail">
            {% endif %}
        </div>
    </div>
</div>
{% endblock %}
```

Code Explanation:

- This template uses Bootstrap to create a responsive form and a clean layout.
- There is a form to edit user information such as username, email, first name, last name, and other profile details, including the profile picture.

7.1.3 Adding URL Routing

Finally, we need to add a URL to access the user profile page. Open the file `users/urls.py` and add the following URL:

```
# File: apps/users/urls.py

from django.urls import path
from . import views

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

Now, users can access their profile page at `/profile/` and view their profile information.

By completing the above steps, you have successfully created a user profile page that allows users to view their information.

Next, we need to ensure that the `users` app URLs are included in the main project routing. Open the file `platform_bot/urls.py` and add the following configuration:

```
# File: platform_bot/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('apps.users.urls')),  # Include
the users app URLs
]
```

With this configuration, the user profile page will be displayed when users visit the root URL of the application.

The profile page includes details such as username, email, first name, last name, address, city, country, postal code, company, about me, and profile picture.

7.2 Creating The Profile Editing Feature

Next, we will create a feature that allows users to edit their information directly on the profile page.

7.2.1 Creating a Form to Edit User Profiles

We have already created the `profile` view in `views.py` to display user information. Now, we will enhance this view to handle both displaying and editing the user's profile.

To allow users to edit their information directly on the profile page without redirecting to a separate page, we will use the `EditProfileForm`. This form combines data from the `User` and `UserProfile` models so users can update their account and profile information on the same page.

Several adjustments can be made to the `EditProfileForm` to ensure that all data is processed correctly. For instance, when saving the data, ensure that changes to the `User` model are also saved.

Here are the adjustments for `EditProfileForm`:

```
# File: apps/users/forms.py

from django import forms
from django.contrib.auth.models import User
from apps.authentication.models import UserProfile

class EditProfileForm(forms.ModelForm):
    # Fields from the User model
    email = forms.EmailField(required=True,
widget=forms.EmailInput(attrs={'class': 'form-
control', 'placeholder': 'Email'}))
    first_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'First Name'}))
```

```
    last_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Last Name'}))

    # Fields from the UserProfile model
    address = forms.CharField(max_length=255,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Address'}))
    city = forms.CharField(max_length=100,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'City'}))
    country = forms.CharField(max_length=100,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Country'}))
    postal_code = forms.CharField(max_length=20,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Postal Code'}))
    company = forms.CharField(max_length=255,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Company'}))
    about = forms.CharField(max_length=255,
required=False, widget=forms.Textarea(attrs={'class':
'form-control', 'placeholder': 'About me', 'rows':
4}))
    foto_profile = forms.ImageField(required=False,
widget=forms.FileInput(attrs={'class': 'form-control-
file'}))

    class Meta:
        model = UserProfile
        fields = ['city', 'address', 'country',
'postal_code', 'about', 'company', 'foto_profile']

    def __init__(self, *args, **kwargs):
        user = kwargs.pop('user', None)  # Retrieve
the user data
        super(EditProfileForm, self).__init__(*args,
**kwargs)
```

```
        if user:
            # Initialize fields from the User model
            self.fields['username'] =
forms.CharField(
                initial=user.username,

widget=forms.TextInput(attrs={'class': 'form-
control', 'readonly': 'readonly'})
            )
            self.fields['email'].initial = user.email
            self.fields['first_name'].initial =
user.first_name
            self.fields['last_name'].initial =
user.last_name

    def save(self, commit=True):
        user = self.instance.user
        user.email = self.cleaned_data['email']
        user.first_name =
self.cleaned_data['first_name']
        user.last_name =
self.cleaned_data['last_name']

        if commit:
            user.save()
            super(EditProfileForm,
self).save(commit=True)
        return user
```

With this configuration, users can edit their information directly on the profile page without needing to navigate to another page.

The form we created, `EditProfileForm`, is quite comprehensive. It combines fields from both the `User` and `UserProfile` models and initializes the `User` fields within the form. However, we need to ensure that the `username` field cannot be edited by users, as it is typically a constant field.

7.2.2 Updating Profile Views

Next, we will update the view that will control the logic for the user's profile page. This view will handle data retrieval, user profile editing, and displaying the user profile.

Open the file `apps/users/views.py` and add the following code:

```
# File: apps/users/views.py
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import import
login_required
from apps.users.forms import EditProfileForm

@login_required
def profile(request):
    user = request.user
    profile = user.userprofile  # Get the profile of
the logged-in user

    if request.method == 'POST':
        form = EditProfileForm(request.POST,
request.FILES, instance=profile, user=user)
        if form.is_valid():
            form.save()
            return redirect('profile')  # Redirect to
the profile page after success
    else:
        form = EditProfileForm(instance=profile,
user=user)

    # Define the context to be passed to the template
    context = {
        'ASSETS_ROOT': '/static/assets',  # Static
assets
        'form': form,  # Form for editing the profile
        'profile': profile,  # Logged-in user's
profile
    }
    # Return the context along with rendering the
page
    return render(request, 'users/profile.html',
context)
```

In the code above, the `profile` function manages the user's profile page view.

7.2.3 Updating the Profile Page Template

Here are some improvements to the `profile.html` template code to make it more compatible with Django forms and ensure that all fields are correctly retrieved from the Django form:

Here are the improvements to our code:

```
<!-- apps/templates/users/profile.html -->
{% extends "dashboard/base.html" %}

{% block content %}
<div class="content">
    <div class="row">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">
                    <h5 class="title">Edit
Profile</h5>
                </div>
                <div class="card-body ">
                    <form method="post"
enctype="multipart/form-data">
                        {% csrf_token %}
                        <div class="row">
                            <div class="col-md-6">
                                <div class="form-
group">

<label>Username</label>
```

```
                                    {{ form.username
}}
                                <div>
                            </div>
                            <div class="col-md-6">
                                <div class="form-
group">

<label>Email</label>
                                    {{ form.email }}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-6">
                                <div class="form-
group">
                                    <label>First
Name</label>

{{ form.first_name }}
                                </div>
                            </div>
                            <div class="col-md-6">
                                <div class="form-
group">
                                    <label>Last
Name</label>
                                    {{ form.last_name
}}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-12">
                                <div class="form-
group">

<label>Company</label>
                                    {{ form.company
}}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-12">
                                <div class="form-
group">

<label>Address</label>
                                    {{ form.address
}}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-4">
                                <div class="form-
group">

<label>City</label>
                                    {{ form.city }}
                                </div>
                            </div>
                            <div class="col-md-4">
                                <div class="form-
group">

<label>Country</label>
                                    {{ form.country
}}
                                </div>
                            </div>
                            <div class="col-md-4">
                                <div class="form-
group">
                                    <label>Postal
Code</label>

{{ form.postal_code }}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-12">
```

```
                                <div class="form-
group">
                                    <label>About
Me</label>
                                    {{ form.about }}
                                </div>
                            </div>
                        </div>
                        <!-- Field for profile
picture -->
                        <div class="form-group">
                            <label>Profile
Picture</label>
                            {{ form.foto_profile }}
                        </div>
                        <button type="submit"
class="btn btn-fill btn-primary">Save</button>
                    </form>
                </div>
            </div>
        </div>
        <div class="col-md-4">
            <div class="card card-user text-center">
                <div class="card-body">
                    <div class="author">
                        <img class="avatar img-fluid"
src="{% if form.foto_profile.value %}
{{ form.foto_profile.value.url }}{% else
%}/static/assets/img/default-avatar.png{% endif %}"
alt="Profile Picture">
                        <h5 class="title text-
center">{{ form.username.value }}</h5>
                    </div>
                    <div class="card-description
text-center">
                        {{ form.about.value }}
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

The improvements made:

- Direct use of `{{ form.field }}` allows Django to handle form elements automatically.
- Removed the separation between `profile_form` and `form`. All fields are now handled by the `EditProfileForm`.
- The `foto_profile` field is directly rendered using `{% if form.foto_profile.value %}` to ensure profile picture handling remains valid.

With these improvements, the form display becomes more consistent with Django, and the editing process can be done on the same page.

7.2.4 Testing the User Profile Page

The first step is to ensure that the user profile page is displayed correctly and can be used to edit profile information.

Access the Profile Page

Open a web browser and log into your application with a registered user account. Navigate to the user profile page. Typically, the URL for the profile page is `/profile/`. Ensure you are directed to the appropriate page.

Verify the Profile Page Display

Check that all expected elements are present on the page, including the form for editing the profile, the profile picture, and user information. Verify that the data displayed in the form matches the data stored in the database.

Test Profile Editing Functionality

Try editing some information in the profile, such as the first name, last name, and email address. Enter the desired changes and click the "Save" button.

```
<!-- users/templates/dashboard/profile.html -->
<input type="text" class="form-control"
name="first_name"
value="{{ form.first_name.value }}">
<input type="email" class="form-control" name="email"
value="{{ form.email.value }}">
```

Upload Profile Picture

Test the profile picture upload by selecting a new image and saving it. Ensure the new image is displayed correctly after saving.

By following these steps, you can ensure that the user profile page functions properly and provides an optimal user experience.

7.3 Integration With Activity Log Model

Next, we need to enhance the display of activity logs on the user dashboard. To do this, we must add logic in the dashboard view as well as improve the dashboard template to show activity logs.

We have already created the `ActivityLog` model in the previous chapter; the next step is to integrate the logging functionality in the view that manages user profile editing. Each time a user edits their profile, that action will be logged.

Open the `views.py` file in the users application and adjust the code for the profile editing view as follows:

```
# File: apps/users/views.py

from django.shortcuts import render, redirect
from django.contrib.auth.decorators import
login_required
from apps.users.forms import EditProfileForm
from apps.dashboard.models import ActivityLog
from django.utils import timezone

@login_required
def profile(request):
    user = request.user
    profile = user.userprofile  # Get the profile of
the logged-in user

    if request.method == 'POST':
        form = EditProfileForm(request.POST,
request.FILES, instance=profile, user=user)
        if form.is_valid():
            form.save()

            # Create activity log when the profile is
updated
            ActivityLog.objects.create(
                user=user,
                action="Edited profile",
                timestamp=timezone.now()
            )

            return redirect('profile')  # Redirect to
the profile page after successful update
```

```
    else:
        form = EditProfileForm(instance=profile,
user=user)

    # Define the context to be passed to the template
    context = {
        'ASSETS_ROOT': '/static/assets',  # Static
assets
        'form': form,  # Form for editing the profile
        'profile': profile,  # Profile of the logged-
in user
    }

    return render(request, 'users/profile.html',
context)
```

The code above adds activity logging each time a user successfully changes their profile. This log will be saved in the `ActivityLog` model, recording the user who made the change, the type of action (in this case, "Edited profile"), and the time of the event.

7.3.1 Improving the Template to Display Activity Logs on the Dashboard

Next, we need to enhance the display of activity logs on the user dashboard. To do this, we must refine the dashboard template to show the activity logs.

The `activity_log` view will display all activity logs for the logged-in user in chronological order, with the most recent logs appearing at the top. Next, modify the `dashboard.html` template file located in the `apps/users/templates/dashboard/` folder:

Let's start by reviewing the HTML code in both files (`base.html` and `dashboard.html`). First, we will ensure that the Bootstrap components used are properly structured to support responsiveness.

Improving `dashboard.html`

The `dashboard.html` file is a dedicated page displayed to users when they log into the dashboard. Let's refine it to make its appearance more responsive and appealing.

```
<!-- File: apps/templates/dashboard/dashboard.html --
>
{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <div class="row">
        <div class="col-lg-12">
            <div class="jumbotron text-center bg-
primary text-white py-5">
                <h2>Welcome to the Dashboard,
{{ user_profile.user.username }}!</h2>
                <p>Here you can manage your bots,
view statistics, and check your latest activity
logs.</p>
            </div>
        </div>
    </div>
    <!-- Display messages if there are errors or
success -->
    {% if messages %}
        <div class="messages">
            {% for message in messages %}
                <div class="alert {{ message.tags
}}">{{ message }}</div>
            {% endfor %}
        </div>
    {% endif %}
    <div class="row mt-5">
        <div class="col-lg-12">
            <h3 class="text-center">Activity
Logs</h3>
            <table class="table table-striped table-
hover">
                <thead class="thead-dark">
                    <tr>
                        <th scope="col">Time</th>
                        <th scope="col">Activity</th>
                    </tr>
                </thead>
                <tbody>
                    {% for log in activity_logs %}
                    <tr>
                        <td>{{ log.timestamp|date:"d-
m-Y H:i" }}</td>
                        <td>{{ log.action }}</td>
                    </tr>
                    {% empty %}
                    <tr>
                        <td colspan="2" class="text-
center">No activity logs found.</td>
                    </tr>
                    {% endfor %}
                </tbody>
            </table>
        </div>
    </div>
</div>
{% endblock %}
```

Improvements:

- ***Use of `py-5` on the Jumbotron***: I added vertical padding (`py-5`) to make the text in the jumbotron look better and less cramped.
- ***Responsive Table***: Bootstrap automatically makes the table more responsive and easier to read on various screen sizes.

By following the steps above, we have successfully displayed the user activity logs on the dashboard when they edit their profile.

By restructuring and refining the `base.html` and `dashboard.html`, we have ensured that the built dashboard is more responsive and accessible on mobile devices.

7.4 Complete Code

We will review all the code for this users application. Below is the complete code.

7.4.1 Forms.py

```
# File: apps/users/forms.py

from django import forms
from django.contrib.auth.models import User
from apps.authentication.models import UserProfile

class EditProfileForm(forms.ModelForm):
    # Fields from the User model
    email = forms.EmailField(required=True,
widget=forms.EmailInput(attrs={'class': 'form-
control', 'placeholder': 'Email'}))
    first_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'First Name'}))
    last_name = forms.CharField(max_length=100,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Last Name'}))

    # Fields from the UserProfile model
    address = forms.CharField(max_length=255,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Address'}))
    city = forms.CharField(max_length=100,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'City'}))
    country = forms.CharField(max_length=100,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Country'}))
    postal_code = forms.CharField(max_length=20,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Postal Code'}))
    company = forms.CharField(max_length=255,
required=False,
widget=forms.TextInput(attrs={'class': 'form-
control', 'placeholder': 'Company'}))
    about = forms.CharField(max_length=255,
required=False, widget=forms.Textarea(attrs={'class':
'form-control', 'placeholder': 'About me', 'rows':
4}))
    foto_profile = forms.ImageField(required=False,
widget=forms.FileInput(attrs={'class': 'form-control-
file'}))

    class Meta:
        model = UserProfile
        fields = ['city', 'address', 'country',
'postal_code', 'about', 'company', 'foto_profile']

    def __init__(self, *args, **kwargs):
        user = kwargs.pop('user', None)  # Get user
data
        super(EditProfileForm, self).__init__(*args,
**kwargs)

        if user:
            # Initialize fields from the User model
            self.fields['username'] =
forms.CharField(
                initial=user.username,

widget=forms.TextInput(attrs={'class': 'form-
control', 'readonly': 'readonly'})
            )
            self.fields['email'].initial = user.email
            self.fields['first_name'].initial =
user.first_name
            self.fields['last_name'].initial =
user.last_name

    def save(self, commit=True):
        user = self.instance.user
        user.email = self.cleaned_data['email']
        user.first_name =
self.cleaned_data['first_name']
        user.last_name =
self.cleaned_data['last_name']

        if commit:
            user.save()
            super(EditProfileForm,
self).save(commit=True)
        return user
```

7.4.2 Views.py

```
# File: apps/users/views.py

from django.shortcuts import render, redirect
from django.contrib.auth.decorators import
login_required
from apps.users.forms import EditProfileForm
from apps.dashboard.models import ActivityLog
from django.utils import timezone

@login_required
def profile(request):
    user = request.user
    profile = user.userprofile  # Get the user
profile of the logged-in user
```

```
    if request.method == 'POST':
        form = EditProfileForm(request.POST,
request.FILES, instance=profile, user=user)
        if form.is_valid():
            form.save()

            # Create activity log when the profile is
updated
            ActivityLog.objects.create(
                user=user,
                action="Edited profile",
                timestamp=timezone.now()
            )

            return redirect('profile')  # Redirect to
the profile page after successful update
    else:
        form = EditProfileForm(instance=profile,
user=user)

    # Define the context to be passed to the template
    context = {
        'ASSETS_ROOT': '/static/assets',  # Static
assets
        'form': form,  # Form for editing profile
        'profile': profile,  # Logged-in user profile
    }

    return render(request, 'users/profile.html',
context)
```

7.4.3 Profile.html

```
<!-- apps/templates/users/profile.html -->
{% extends "dashboard/base.html" %}

{% block content %}
<div class="content">
    <div class="row">
        <div class="col-md-8">
```

```
            <div class="card">
                <div class="card-header">
                    <h5 class="title">Edit
Profile</h5>
                </div>
                <div class="card-body ">
                    <form method="post"
enctype="multipart/form-data">
                        {% csrf_token %}
                        <div class="row">
                            <div class="col-md-6">
                                <div class="form-
group">

<label>Username</label>
                                    {{ form.username
}}
                                </div>
                            </div>
                            <div class="col-md-6">
                                <div class="form-
group">

<label>Email</label>
                                    {{ form.email }}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-6">
                                <div class="form-
group">
                                    <label>First
Name</label>

{{ form.first_name }}
                                </div>
                            </div>
                            <div class="col-md-6">
                                <div class="form-
group">
                                    <label>Last
Name</label>
```

```
                                    {{ form.last_name
}}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-12">
                                <div class="form-
group">

<label>Company</label>
                                    {{ form.company
}}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-12">
                                <div class="form-
group">

<label>Address</label>
                                    {{ form.address
}}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-4">
                                <div class="form-
group">

<label>City</label>
                                    {{ form.city }}
                                </div>
                            </div>
                            <div class="col-md-4">
                                <div class="form-
group">

<label>Country</label>
                                    {{ form.country
}}
```

```
                                </div>
                            </div>
                            <div class="col-md-4">
                                <div class="form-
group">
                                    <label>Postal
Code</label>

{{ form.postal_code }}
                                </div>
                            </div>
                        </div>
                        <div class="row">
                            <div class="col-md-12">
                                <div class="form-
group">
                                    <label>About
Me</label>
                                    {{ form.about }}
                                </div>
                            </div>
                        </div>
                        <!-- Field for profile
picture -->
                        <div class="form-group">
                            <label>Profile
Picture</label>
                            {{ form.foto_profile }}
                        </div>
                        <button type="submit"
class="btn btn-fill btn-primary">Save</button>
                    </form>
                </div>
            </div>
        </div>
        <div class="col-md-4">
            <div class="card card-user text-center">
                <div class="card-body">
                    <div class="author">
                        <img class="avatar img-fluid"
src="{% if form.foto_profile.value %}
{{ form.foto_profile.value.url }}{% else
%}/static/assets/img/default-avatar.png{% endif %}"
alt="Profile Picture">
                        <h5 class="title text-
center">{{ form.username.value }}</h5>
                    </div>
                    <div class="card-description
text-center">
                        {{ form.about.value }}
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

7.4.4 URLS.py

```
# File: apps/users/urls.py

from django.urls import path
from . import views

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

With all this code, it is expected that there will be no errors in the application, ensuring a smooth and functional user experience.

Chapter Conclusion

In this chapter, we thoroughly discussed how to manage user profiles in a Django application, focusing on creating a profile page, features for editing information, and integrating with activity logs.

Section 7.1 began with the creation of the user profile page, which is a crucial element for providing users access to their personal information. We explained how to create a view for the profile page, design an appropriate template, and set up a URL to access the page. This process ensures that users can easily view and manage their personal information.

In **Section 7.2**, we covered the features for editing user information. This part includes creating a form to edit the profile, updating the profile views to handle the updated data, and adjusting the profile page template to accommodate the changes. Testing the profile page was also included to ensure that the editing features function correctly and provide a smooth user experience.

Section 7.3 focused on integrating with the activity log model, which is essential for tracking and displaying user activity within the application. We discussed how to refine the template to show activity logs on the dashboard and how to create a responsive sidebar using Bootstrap for better navigation.

Overall, this chapter provides a comprehensive guide on user profile management, from creating a basic profile page to adding editing features and integrating activity logs. By implementing these steps, you can build a robust and user-friendly profile management system that allows users to effectively manage their information and clearly view their activity history.

Chapter 8 - Introduction to Bots and Setting Up the Basic Page

In this section, we will begin creating the page for bot management. However, it is important to note that at this stage, the page does not yet have full functionality; instead, we will be laying the basic foundation. This foundation includes the models and forms that we will use to manage bots, such as creating new bots and viewing the list of existing bots.

The model we will create will reflect the data structure of the bots that we will store in the database. Here, we will also prepare a form that will be used to input data when users create or manage bots.

8.1 Setting Up Models And Forms For Bots

Let’s start by setting up the Bot model that will be used to store bot data in the database. This model will be placed in the file `apps/bots/models.py`. After that, we will prepare a simple form to manage that data.

8.1.1 Setting Up the Bot Model

At this stage, we will discuss the model used to store bot data within the Django application. This model is crucial as it holds key information such as the type of bot, unique token, webhook URL, automated messages, and bot status. By understanding the components of this model, we can manage the various bots created by users on our platform, whether they are Telegram or WhatsApp bots.

The Bot model will be created in the file `apps/bots/models.py`. In this file, we will define the Bot model with the necessary attributes, such as the bot type, token, status, and automated messages.

The first step is to open the `models.py` file in the `apps/bots/` folder. We start by importing the required modules:

```python
# File: apps/bots/models.py
from django.db import models
from django.contrib.auth.models import User
```

In this line, we import `models` from `django.db` to define the model, and `User` from `django.contrib.auth.models`, which allows us to link the created bot with a specific user. The relationship between bots and users is very important so that each bot created can be associated with a valid user account.

Next, we define the `Bot` class, which will represent the basic model for each bot. In this model, we will include attributes such as name, description, token, webhook URL, and automated messages.

```
# File: apps/bots/models.py

class Bot(models.Model):
    BOT_CHOICES = [
        ('telegram', 'Telegram'),
        ('whatsapp', 'WhatsApp'),
    ]
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # The bot will be
associated with a user
    bot_type = models.CharField(max_length=20,
choices=BOT_CHOICES)  # Type of bot: Telegram or
WhatsApp

    token_telegram = models.CharField(max_length=255,
unique=True, help_text="Enter a unique bot token.")
# Telegram token
    webhook_url = models.URLField(max_length=255,
blank=True, help_text="The bot's webhook URL will be
set automatically.")  # Webhook URL

    is_active = models.BooleanField(default=True,
help_text="Bot status: active or not.")  # Active or
inactive status

    # Automated messages
    start_message = models.TextField(blank=True,
default="Welcome to the bot!", help_text="Message
sent when the user types /start.")
    help_message = models.TextField(blank=True,
default="Here’s how to use the bot.",
help_text="Message sent when the user types /help.")
```

```
    messages = models.JSONField(default=list,
blank=True, help_text="List of received messages and
their responses in JSON format.")  # JSON format

    # Timestamps
    created_at =
models.DateTimeField(auto_now_add=True)  # Date the
bot was created
    updated_at = models.DateTimeField(auto_now=True)
# Date the bot was last updated

    name = models.CharField(max_length=255,
blank=True, default="My Bot", help_text="Bot name.")
# Bot name
    description = models.TextField(blank=True,
default="This is my first bot", help_text="Bot
description.")  # Bot description

    def __str__(self):
        return f'{self.bot_type.capitalize()} Bot -
{self.user.username}'

    class Meta:
        verbose_name = "Bot"
        verbose_name_plural = "Bots"
```

Complete Code

```
# File: apps/bots/models.py

from django.db import models
from django.contrib.auth.models import User

class Bot(models.Model):
    BOT_CHOICES = [
        ('telegram', 'Telegram'),
        ('whatsapp', 'WhatsApp'),
    ]
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # The bot will be
associated with a user
```

```
    bot_type = models.CharField(max_length=20,
choices=BOT_CHOICES)  # Type of bot: Telegram or
WhatsApp

    token_telegram = models.CharField(max_length=255,
unique=True, help_text="Enter a unique bot token.")
# Telegram token
    webhook_url = models.URLField(max_length=255,
blank=True, help_text="The bot's webhook URL will be
set automatically.")  # Webhook URL

    is_active = models.BooleanField(default=True,
help_text="Bot status: active or not.")  # Active or
inactive status

    # Automated messages
    start_message = models.TextField(blank=True,
default="Welcome to the bot!", help_text="Message
sent when the user types /start.")
    help_message = models.TextField(blank=True,
default="Here’s how to use the bot.",
help_text="Message sent when the user types /help.")

    messages = models.JSONField(default=list,
blank=True, help_text="List of received messages and
their responses in JSON format.")  # JSON format

    # Timestamps
    created_at =
models.DateTimeField(auto_now_add=True)  # Date the
bot was created
    updated_at = models.DateTimeField(auto_now=True)
# Date the bot was last updated

    name = models.CharField(max_length=255,
blank=True, default="My Bot", help_text="Bot name.")
# Bot name
    description = models.TextField(blank=True,
default="This is my first bot", help_text="Bot
description.")  # Bot description

    def __str__(self):
        return f'{self.bot_type.capitalize()} Bot -
{self.user.username}'

    class Meta:
        verbose_name = "Bot"
        verbose_name_plural = "Bots"
```

Code Explanation:

- ***user***: This attribute is a ForeignKey relationship between the Bot model and the User model. Each bot is associated with one user, and with the parameter `on_delete=models.CASCADE`, if the user is deleted, all bots owned by that user will also be deleted.
- ***bot_type***: This column uses options defined by `BOT_CHOICES` to determine whether the bot is a Telegram bot or a WhatsApp bot.
- ***token_telegram***: For Telegram bots, we require a unique token. This token is essential as it is used to identify and control the bot on the Telegram platform.
- ***webhook_url***: The webhook URL is the endpoint used by the bot to receive messages from the platform (Telegram or WhatsApp). This URL can be left blank as it will be set automatically based on the configuration.
- ***is_active***: This attribute indicates whether the bot is active or not. If true, the bot will function normally. If false, the bot is considered inactive.
- ***start_message and help_message***: These two columns serve as automated messages sent when the user issues the /start and /help commands to the bot. These messages can be customized as desired by the user.

- ***messages***: This column stores a list of messages and their responses in JSON format. The JSON format makes it easier for us to organize automated responses based on messages received from users.
- ***created_at and updated_at***: These two attributes log when the bot was created (`created_at`) and when it was last updated (`updated_at`). These values will be automatically set by Django.

After understanding this model, we have laid the basic structure for storing and managing bots within our application. This model also prepares us for the next steps, which are creating the forms and views necessary to add and manage bots through the user interface.

Next Steps: Creating and Applying Migrations

Once the Bot model is defined, the next step is to create and apply migrations to reflect these changes in the database.

Applying migrations ensures that all the columns and relationships we have defined will be created in the appropriate database tables, allowing bot data to be stored correctly.

```
$ python manage.py makemigrations
$ python manage.py migrate
```

With this step, the Bot model is now ready for use in our application. We can now proceed to create the forms and views for adding and managing bots.

8.1.2 Setting Up the Form for the Bot

After the bot model has been defined, the next step in the bot application development process is to set up the form that will be used to manage bot data. This form will allow users to input the necessary information to create or edit a bot.

At this stage, we will open the file `apps/bots/forms.py` and start defining the form that will help facilitate the data input process from users. By using the `ModelForm` provided by Django, we can easily create a form that is directly connected to the model we created earlier.

First, we need to import the `forms` module from Django as well as the `Bot` model that we defined earlier:

```
# File: apps/bots/forms.py

from django import forms  # Importing the forms
module from Django
from .models import Bot  # Importing the Bot model
that was created earlier
```

In the code above, we start by importing `forms` from Django, which will assist us in defining the form. Additionally, we also import the `Bot` model from the models file that we created in the previous stage.

Creating the BotForm Class

The next step is to create the `BotForm` class that will handle the creation and updating of the bot. This class will use Django's `ModelForm` to connect the form with the `Bot` model. The form will include input for basic information such as the type of bot (Telegram or WhatsApp). Here is the complete code:

```
# File: apps/bots/forms.py

from django import forms
from .models import Bot  # Importing the Bot model

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot  # Connecting the form to the Bot
model
        fields = ['bot_type', 'token_telegram',
'name', 'start_message', 'help_message',
'description', 'is_active']  # Fields displayed in
the form
        widgets = {
            'name': forms.TextInput(attrs={'class':
'form-control', 'placeholder': 'Enter bot name'}),
            'description':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Bot description'}),
        }
        help_texts = {
            'token_telegram': 'Enter the unique token
received from the Telegram platform.',
            'name': 'The bot name can be changed as
desired.',
            'description': 'A brief description of
the bot and its functions.',
        }
```

In the code above, we utilize several important features of `ModelForm` to make the data entry process easier for users:

- ***Meta***: This section connects the form with the `Bot` model and defines which fields will be displayed in the form, such as the type of bot (`bot_type`), token, name, description, and the bot's status (`is_active`).
- ***Widgets***: With widgets, we can control the appearance of each field in the form. For example, we add `class='form-control'` for the name and description elements, making the form look neater by utilizing Bootstrap's built-in classes.
- ***Help Texts***: To guide users, we also add `help_texts` that provide additional explanations for certain fields, such as the Telegram token and bot description, helping users better understand what data they need to input.

Code Explanation

The code above prepares a form that helps users manage their bots. A brief explanation of each component of this form is as follows:

- ***Meta***: Connects the form to the `Bot` model and specifies which fields will be included in the form. In this case, the displayed fields are the type of bot, token, name, description, and the bot's status.
- ***Widgets***: This section alters the appearance of the input fields. For example, `name` uses `TextInput` with the additional attribute `class='form-control'` to make it look more like a Bootstrap component, ensuring a more consistent and user-friendly appearance. The

`description` field uses `Textarea` with additional rows.

- ***Help Texts***: Provides additional information to users to better understand each field on the form. This helps ensure that users do not get confused when filling out fields that may be less familiar, such as `token_telegram`.

Connecting the Form with the View

Once the form has been successfully created, the next step is to connect this form with our view, allowing the data filled in by the user to be saved in the database. However, for now, we will focus on preparing the form, and the view implementation will be explained in the following subsection.

With the form created, users can easily fill in the necessary information to create or edit a bot. This provides a solid foundation for more complex bot management in the future, such as connecting the bot to a webhook or adding more advanced management features.

In the next stage, we will implement the logic behind this form in the view so that the data entered by users can be processed and saved correctly in the database.

8.2 Setting Up The Bot Handler

After we have prepared the form to manage the bot, the next step is to set up the functions that will handle the interactions between the bot and the users. In this section, we will implement handlers for both the Telegram and WhatsApp bots. These handlers serve as a link between the messages received by the bot and the responses that will be sent back to the users. In other words, the handler is responsible for processing user requests and providing appropriate answers.

Create a folder named `services` inside the `bots` folder, and create files `telegram.py` and `whatsapp.py` to place the `handle_update` functions inside them.

```
bots/
    views.py
    services/
        telegram.py
        whatsapp.py
```

Let’s start by creating handlers for Telegram and WhatsApp in the files `apps/bots/services/telegram.py` and `apps/bots/services/whatsapp.py`.

8.2.1 Handle Telegram

After we set up the models and forms for the bot, the next step is to prepare the services that will handle the interactions between the users and the bot. In this case, for the Telegram bot, we need to create a mechanism that can receive and process updates from users.

When a user sends a message to the bot, Telegram will send data in JSON format containing information about the message. Our task is to process this data and send back the appropriate response to the user.

To achieve this, we will utilize the Telegram API, which allows us to retrieve messages, process them, and send the necessary responses. Here, we will discuss the code that handles updates for the Telegram bot. This code will be placed in the file `apps/bots/services/telegram.py`, which will handle all processing logic for the Telegram bot.

Importing Dependencies

We begin by importing the dependencies needed to connect our application with the Telegram API and also to manage the bots registered in the application. In this file, we will use `requests` to make HTTP requests, and the `logging` module to log errors or information during the bot's operation.

```
# File: apps/bots/services/telegram.py

from apps.bots.models import Bot  # Importing the Bot
model from the bots application
import requests  # To send HTTP requests to the
Telegram API
import logging  # For logging errors and information
related to the bot
```

```
logger = logging.getLogger(__name__)  # Initializing
the logger for logging errors
```

In this section, we initialize a logger with a name corresponding to the current module. This logger will be very useful for monitoring if there are any errors in sending messages or when the Telegram API does not respond as expected.

The `handle_update` Function

The `handle_update` function is the main function that will handle updates from the Telegram bot. Whenever a user sends a message to the bot, Telegram will send update data in JSON format, which will then be processed by this function. Here, we will process the message data and determine what action the bot should take based on the commands given by the user.

```
def handle_update(update):
    chat_id = update['message']['chat']['id']  #
Getting the chat_id from the received message
    text = update['message']['text'].lower()  #
Converting the message text to lowercase for easier
comparison

    bot =
Bot.objects.filter(bot_type='telegram').first()  #
Retrieving the first Telegram bot from the database

    if bot:
        # Processing commands received from the user
        if text == "/start":
            response_text = bot.start_message  #
Retrieving the welcome message from the Bot model
        elif text == "/help":
            response_text = bot.help_message  # 
Retrieving the help message from the Bot model
        else:
            response_text = 'Unknown command. Type 
/help for a list of commands.'  # Default message for 
unrecognized commands

        # Sending response message to the user
        send_message(bot.token_telegram, 
"sendMessage", {
            'chat_id': chat_id,
            'text': response_text
        })
```

In this function, we retrieve the `chat_id` from the received message, which is a unique identifier of the conversation between the bot and the user. Then, we get the text from the received message and convert it to lowercase for consistency when compared with existing commands.

After that, we query the database to retrieve the registered Telegram bot, represented by the `Bot` model. If the bot is found, we check the content of the user's message and determine the appropriate response based on commands like `/start` or `/help`. If the given command is not recognized, the bot will respond with a default message.

The `send_message` Function

The `send_message` function is used to send a reply message to the user through the Telegram API. This function requires the bot token stored in the `Bot` model, as well as additional data such as `chat_id` and the message text to be sent.

```
def send_message(token_telegram, method, data):
    telegram_api_url =
f'https://api.telegram.org/bot{token_telegram}/{metho
d}'  # Constructing the URL for the Telegram API
    response = requests.post(telegram_api_url,
data=data)  # Sending POST request to the Telegram
API

    # Checking if the request was successful
    if response.status_code != 200:
        logger.error(f"Failed to send message:
{response.text}")  # Logging error if sending the
message fails
    return response  # Returning the response from
the API
```

This function first builds the Telegram API URL using the bot token and the desired method (in this case, `sendMessage`).

Once the URL is constructed, we send a POST request using `requests.post`, with the required data such as `chat_id` and message text. If the Telegram API responds with a status other than 200 OK, we log the error using `logger.error`, making it easier for us to monitor any issues that may occur when the bot sends messages.

Complete Code

```
# File: apps/bots/services/telegram.py

from apps.bots.models import Bot  # Importing the Bot
model from the bots application
import requests  # To send HTTP requests to the
Telegram API
```

```
import logging  # For logging errors and information
related to the bot

logger = logging.getLogger(__name__)  # Initializing
the logger for logging errors

def handle_update(update):
    chat_id = update['message']['chat']['id']  #
Getting the chat_id from the received message
    text = update['message']['text'].lower()  #
Converting the message text to lowercase for easier
comparison

    bot =
Bot.objects.filter(bot_type='telegram').first()  #
Retrieving the first Telegram bot from the database

    if bot:
        # Processing commands received from the user
        if text == "/start":
            response_text = bot.start_message  #
Retrieving the welcome message from the Bot model
        elif text == "/help":
            response_text = bot.help_message  #
Retrieving the help message from the Bot model
        else:
            response_text = 'Unknown command. Type
/help for a list of commands.'  # Default message for
unrecognized commands

        # Sending response message to the user
        send_message(bot.token_telegram,
"sendMessage", {
            'chat_id': chat_id,
            'text': response_text
        })

def send_message(token_telegram, method, data):
    telegram_api_url =
f'https://api.telegram.org/bot{token_telegram}/{metho
d}'  # Constructing the URL for the Telegram API
```

```
    response = requests.post(telegram_api_url,
data=data)  # Sending POST request to the Telegram
API

    # Checking if the request was successful
    if response.status_code != 200:
        logger.error(f"Failed to send message:
{response.text}")  # Logging error if sending the
message fails
    return response  # Returning the response from
the API
```

In this section, we have established a solid foundation for handling interactions between users and the Telegram bot. By using the `handle_update` function, we can process messages from users and respond appropriately. The `send_message` function helps us in sending reply messages through the Telegram API, as well as logging errors in case of failures.

The next step is to integrate these services more deeply into the application flow, such as connecting with webhooks, so that every update received by the bot can be automatically processed by the functions we have created. With this foundation, we are one step closer to implementing a fully operational bot that users can interact with through the Telegram platform.

8.2.2 Handle WhatsApp

After preparing the handling function for the Telegram bot, the next step is to implement a similar function for WhatsApp. In this case, we will use Twilio's service to send and receive messages via WhatsApp. Twilio provides an API that allows us to manage communication between users and the bot easily. Like the Telegram bot, this function will handle messages received from users, process them, and respond accordingly.

We will use the file `apps/bots/services/whatsapp.py` to handle updates coming from WhatsApp. Here is the code implementation to handle incoming messages:

```
# File: apps/bots/services/whatsapp.py

from django.http import HttpResponse  # To return
HTTP response
from django.views.decorators.csrf import csrf_exempt
# To allow view access without CSRF validation
from twilio.twiml.messaging_response import
MessagingResponse  # To create a response to the
Twilio API
from apps.bots.models import Bot  # Importing the Bot
model from the bots app
import logging  # For logging activities and errors

logger = logging.getLogger(__name__)  # Initialize
logger

@csrf_exempt  # Decorator to disable CSRF validation
on this view
def handle_update(update):
    # Get the message sender and the message content
    user = update['From']
    message = update['Body'].strip().lower()  #
Convert the message to lowercase for comparison

    # Logging the received message
    logger.info(f'{user} says {message}')

    # Retrieve the first WhatsApp bot from the
database
    bot =
Bot.objects.filter(bot_type='whatsapp').first()
```

```
    # Create a response using Twilio's
MessagingResponse
    response = MessagingResponse()

    if bot:  # If the bot is found
        # Check the command sent by the user
        if message == "/start":
            response_text = bot.start_message  #
Welcome message
        elif message == "/help":
            response_text = bot.help_message  # Help
message
        else:
            response_text = 'Unknown command. Type
/help to see the list of commands.'  # Default
message

        response.message(response_text)  # Send the
response message
    else:
        response.message('Bot not found.')  # If no
bot is found in the database

    # Return the response as an HTTP response
    return HttpResponse(str(response))
```

Code Explanation:

- ***Importing Dependencies***
  At the beginning of the code, we import several important modules. `HttpResponse` is used to send the HTTP response back to Twilio, while `csrf_exempt` allows us to disable CSRF validation on this view since Twilio does not send a CSRF token in its webhook. We also import `MessagingResponse` from Twilio to build response messages to users, as well as importing

the `Bot` model to get bot-related information from the database.

- ***handle_update(update)***
  The `handle_update` function is the core of the message handling mechanism in WhatsApp. This function receives the `update` parameter, which contains the JSON data sent by Twilio every time there is an interaction from a user. This data includes information such as the sender's number (`From`) and the message (`Body`). We convert the received message to lowercase using `message.lower()` for easier comparison.
- ***Retrieving Bot from Database***
  In this code, we query the database to get the first bot of type WhatsApp. This logic is similar to what we used in handling Telegram, where we only retrieve one bot first. However, in future developments, we can expand this logic to support multiple WhatsApp bots in one application.
- ***Processing Messages***
  The received message is then checked to see if it contains specific commands, such as `/start` or `/help`. If the user sends the `/start` command, the bot will respond with the welcome message stored in the `start_message` column of the `Bot` model. Similarly, if the `/help` command is received, the bot will send the prepared help message. If the sent command is unknown, the bot will provide a default response suggesting the user type `/help`.

- ***Sending Responses via Twilio API***
  To send a reply to the user, we use `MessagingResponse` from Twilio. This function builds the message sent back to the WhatsApp user via Twilio. After all processes are complete, we return the response as an `HttpResponse` so that Twilio can receive the message and display it to the user.

8.2.3 Platform Handling Function

Every bot platform, whether it's Telegram or WhatsApp, has different data formats and events in how messages are sent through webhooks. Therefore, it is essential to customize the handling functions for each platform according to the data format they send.

In this case, we use the Telegram API for the Telegram bot and the Twilio API for the WhatsApp bot.
Some points to note when handling different platforms include:

- ***Telegram***: Telegram sends data in JSON format, which includes information about messages, users, and chats. We need to ensure that all this data is processed correctly through the `handle_telegram_webhook` function.
- ***WhatsApp***: Data from WhatsApp via Twilio also includes text messages, media, and information about the sender. Therefore, we must handle this format properly in the `handle_whatsapp_webhook` function.

This way, we can effectively manage the data sent by each platform and provide appropriate responses to users. Make sure to always test each bot platform implementation thoroughly to ensure that the bot can respond to messages correctly and according to the desired functionality.

8.3 Setting Up The Page To Create Bots

In this section, we will build a page that allows users to create new bots. These steps will include setting up the view, template, and URL routing so that the bot creation process can be accessed by users through the web page.

Previously, we have set up the model and form for the bot. Now, we will proceed with the implementation of the view, template, and URL routing for the bot creation page.

8.3.1 Creating a View to Create a Bot

At this stage, we will create a view that handles the logic behind the bot creation page. The view is responsible for processing requests from users, whether to display the bot creation form or to process the data submitted by users when the form is submitted. This view will provide the initial foundation for the bot creation process, although at this stage, we have not yet added webhook features or activity logging. These features will be discussed further in upcoming chapters.

We will set up a simple view that allows users to create a bot without integration with webhook or activity log. Although its functionality is limited, this page is already accessible and usable by users. The following code will be placed in the `apps/bots/views.py` file:

```
# File: apps/bots/views.py

from django.shortcuts import render, redirect,
get_object_or_404  # For rendering templates and
redirecting pages
from django.contrib.auth.decorators import
login_required  # To ensure that only logged-in users
can access this view
from .forms import BotForm  # Importing BotForm for
bot creation
from .models import Bot  # Importing Bot model from
models.py

@login_required  # Decorator to ensure only
registered users can access this page
def create_bot(request):
    """
    View for creating a new bot. Webhook and activity
log features have not been integrated yet.
    """
    if request.method == 'POST':  # Checking if the
request is a POST, meaning the user has submitted the
form
        form = BotForm(request.POST)  # Creating an
instance of BotForm with the submitted data
        if form.is_valid():  # Validating the data
submitted through the form
            bot = form.save(commit=False)  # Creating
a Bot instance without immediately saving it to the
database
            bot.user = request.user  # Saving the
information of the user creating the bot
            bot.save()  # Saving the bot to the
database
```

```
            # At this stage, there is no integration
with webhook or activity logging.
            # These features will be discussed in the
next chapter.

            return redirect('manage_bots')  # After
successfully saving the bot, the user is redirected
to the bot management page
    else:
        form = BotForm()  # If the request is GET,
display an empty form for the user to fill out

    return render(request, 'bots/create_bot.html',
{'form': form})  # Render the create_bot page with
the form
```

Code Explanation:

- ***Importing Modules***
  At the beginning of the code, we import several necessary modules. The `render` function is used to display the HTML template, while `redirect` directs the user to another page after the bot is successfully created. The `get_object_or_404` function will be useful if we need to retrieve an existing object from the database based on specific parameters, but in this example, it hasn't been used. Next, `login_required` ensures that only logged-in users can access this view. We also import `BotForm`, which is used as the form for creating the bot, and the `Bot` model that stores bot data.
- ***Function create_bot(request)***
  The `create_bot` function is the core of the bot creation logic. This function handles two main types of requests: GET requests to display an empty form and

POST requests to process the data filled out by users in the form.

- ***Handling POST Method***
  When the user fills out the form and submits it, the request method changes to POST. In this case, the view will process the submitted data using `BotForm(request.POST)`. After that, the data is validated using `form.is_valid()`. If the validation is successful, a bot object is created from the form with `commit=False`, meaning the bot has not yet been saved to the database. Next, the user information creating the bot is stored in the `user` property of the bot object. Then, the bot is saved to the database using `bot.save()`.
- ***Redirecting to the Bot Management Page***
  After the bot is successfully saved, the user is redirected to the bot management page through `redirect('manage_bots')`. This redirection provides a smooth and intuitive flow for users, as they will be taken directly to the page displaying the bots they have created.
- ***Displaying an Empty Form (GET Method)***
  If the user accesses this page without submitting a form (via a GET request), the view will display an empty form using `form = BotForm()`. This allows the user to fill out the form and start the bot creation process.
- ***Template Rendering***
  The `render` function is used to display the

`create_bot.html` page with the prepared form. This form can later be filled out by users to create a new bot. At this stage, although the webhook and activity logging functionalities are not yet integrated, the basic structure of the bot creation page is ready and can be used.

In this section, we have set up the bot creation page that functions to display the form and process the data submitted by users. Although this page does not yet have webhook and activity log features, we have built a foundational structure for displaying the form and saving the newly created bot to the database.

Integration with webhook and activity logging will be discussed further in the next chapter.

8.3.2 Creating a Template for Bot Creation

After setting up the view that handles the logic for creating bots, the next step is to create a template that will serve as the user interface. This template is responsible for displaying the form we created in the previous view and providing elements that can be filled in by the user before the data is sent to the server. In Django, templates not only display static information but can also utilize variables and dynamic components sent by the view.

In this case, we will create a template called `create_bot.html`, which will be the bot creation page. Users can choose the type of bot they want to create (Telegram or WhatsApp), fill in the necessary information according to the selected platform, and then submit the data. Although at this stage the template does not have webhook functionality or activity logging, it will work well for the basic bot creation process.

This template acts as a user interface that allows direct interaction with the form. The template will display the bot creation form, provide basic validation through the use of CSRF tokens, and utilize some JavaScript functions to enhance user experience, such as hiding or displaying fields based on the chosen platform.

Below is the code for the `create_bot.html` template, which will be placed in the `templates/bots/create_bot.html` directory:

```
<!-- File: apps/templates/bots/create_bot.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <!-- Card for Form -->
    <div class="card">
        <div class="card-header">
            <h5 class="card-title">Create New
Bot</h5>
        </div>
        <div class="card-body">
            <!-- Form for Bot -->
            <form method="post">
                {% csrf_token %}
                <!-- Form Group for Platform (appears
first) -->
                <div class="form-group">
                    <label class="form-label">
                        Platform
                    </label>
                    <div class="btn-group"
role="group" aria-label="Platform">
                        <input type="radio"
name="bot_type" id="telegram" value="telegram" {% if
form.bot_type.value == 'telegram' %}checked{% endif
%} onclick="toggleFields()">
                        <label
for="telegram">Telegram</label>

                        <input type="radio"
name="bot_type" id="whatsapp" value="whatsapp" {% if
form.bot_type.value == 'whatsapp' %}checked{% endif
%} onclick="toggleFields()">
                        <label
for="whatsapp">WhatsApp</label>
                    </div>
                </div>

                <!-- Form Group for Name -->
                <div class="form-group">
                    <label
for="{{ form.name.id_for_label }}">
                        Name
                    </label>
                    <input type="text" name="name"
id="{{ form.name.id_for_label }}"
value="{{ form.name.value }}" class="form-control {%
if form.name.errors %}is-invalid{% endif %}">
                    {% for error in form.name.errors
%}
                        <div class="invalid-
feedback">{{ error }}</div>
                    {% endfor %}
                </div>
                <!-- Form Group for Token (only for
Telegram) -->
                <div class="form-group" id="token-
field">
                    <label
for="{{ form.token_telegram.id_for_label }}">
                        Token
                    </label>
                    <input type="text" name="token"
id="{{ form.token_telegram.id_for_label }}" value="{{
form.token_telegram.value }}" class="form-control {%
if form.token_telegram.errors %}is-invalid{% endif
%}">
                    {% for error in
form.token_telegram.errors %}
                        <div class="invalid-
feedback">{{ error }}</div>
                    {% endfor %}
                </div>

                <!-- Submit Button -->
                <button type="submit" class="btn btn-
primary">
                    Create Bot
                </button>
            </form>
        </div>
    </div>

    <script>
        // Function to hide/show elements based on
platform
        function toggleFields() {
            var tokenField =
document.getElementById('token-field');
            var whatsappFields =
document.getElementById('whatsapp-fields');

            if
(document.getElementById('telegram').checked) {
                tokenField.classList.remove('d-
none');
```

```
                whatsappFields.classList.add('d-
none');
            } else if
(document.getElementById('whatsapp').checked) {
                tokenField.classList.add('d-none');
                whatsappFields.classList.remove('d-
none');
            }
        }

        // Call toggleFields() when the page loads
        toggleFields();
    </script>
{% endblock %}
```

Code Explanation:

- ***Using Base Template***
  This template extends the base template `base.html`, which means it inherits the previously defined page structure. This makes it easier to manage a consistent layout across all pages of our application. All content specific to the bot creation page will be placed within the `{% block content %}`.
- ***Bot Creation Form***
  This form is designed using simple HTML elements styled with the Bootstrap framework to look more professional and responsive. Users can select the bot platform they want to create (Telegram or WhatsApp) from radio buttons. Once a platform is selected, the relevant fields will appear. For instance, if the user selects Telegram, the field for entering the Telegram token will be displayed; whereas if WhatsApp is selected, the relevant fields for WhatsApp (such as

Account SID, Auth Token, and WhatsApp number) will appear.

- ***Using CSRF Token***
  The CSRF token (`{% csrf_token %}`) is an important security element in every form in Django. This token helps prevent CSRF (Cross-Site Request Forgery) attacks by ensuring that only valid forms can submit data.
- ***Simple Validation with JavaScript***
  This template uses a simple JavaScript function, `toggleFields()`, to determine which fields to display based on the platform selected by the user. When the user selects Telegram, the Telegram token field will appear, and when they choose WhatsApp, the relevant WhatsApp fields will appear. This logic is implemented by showing or hiding HTML elements using the display property.
- ***Submit Button***
  After the user fills in all the required information, they can submit the form by pressing the "Create Bot" button. This form will be sent to the server, and the data filled in by the user will be processed by the view we prepared earlier.

In this section, we have created the `create_bot.html` template, which serves as the bot creation interface. This template allows users to select the platform they will use (Telegram or WhatsApp) and fill in the necessary information, such as tokens or other credentials. By using simple JavaScript, we can create an interactive and dynamic interface, so users only see the fields relevant to the platform they choose. This template is also equipped with a security mechanism through the CSRF token, ensuring that the form can only be accessed and used by valid users.

At this stage, we have successfully built the foundation of a secure and intuitive bot creation interface. More advanced functionalities such as webhook integration and activity logging will be continued in the next chapter.

8.3.3 Setting Up URL Routing for Bot Creation

The final step is to configure the URL routing so that users can access the bot creation page through the appropriate URL in our web application.

Open the `apps/bots/urls.py` file and add the following route:

```
# File: apps/bots/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('create/', views.create_bot,
name='create_bot'),  # URL for the bot creation page
]
```

Code Explanation:

- **`path('create/', views.create_bot, name='create_bot')`:** This URL connects the bot creation page with the view we created earlier (`create_bot`). Users can access this page via `/create/`.

Next, we need to ensure that the `bots` app URLs are included in the main project routing. Open the `platform_bot/urls.py` file and add the following configuration:

```
# File: platform_bot/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('apps.bots.urls')),  # Include
the bots app URLs
]
```

With this configuration, the bot pages will be accessible when users visit the root URL of the application.

At this point, the bot creation page is now accessible, and users can fill out the bot creation form we've set up.

In this section, we've successfully built a page for bot creation, complete with the view, template, and URL routing. Although the bot creation functionality isn't fully complete (as it will require further handling, such as webhooks), the foundation for managing bot creation is in place.

8.4 Creating The Bot Management Page

In this section, we will create a management page where users can view a list of bots they've created and perform actions such as deleting a bot. Similar to the previous section, we'll set up the view, template, and URL routing for this page, allowing users to easily manage their bots.

This page will serve as a control center where users can manage the bots they've created. Here, they can see a list of bots, delete those that are no longer needed, and perform other actions in the future.

8.4.1 Creating the Bot Management View

After creating the bot creation template, the next step is to provide a page where users can manage the bots they've created. In this process, we'll create a view that displays a list of bots owned by the user and handles actions like deleting a bot.

This view will fetch the bots owned by the logged-in user, display them in a table or card format, and allow users to perform actions such as deleting a bot. Additionally, the view needs to be equipped with security measures to prevent unauthorized access or deletion of bots by other users. For this, we will use several

Django features, such as `login_required` and `get_object_or_404`.

First, open the `views.py` file inside the `apps/bots/` directory. Here, we will write the view for the bot management page. Make sure the file is properly set up, and add the following code:

```
# File: apps/bots/views.py

from django.contrib.auth.decorators import
login_required
from django.shortcuts import render,
get_object_or_404, redirect
from .models import Bot

@login_required  # Ensure only logged-in users can
access this view
def manage_bots(request):
    bots = Bot.objects.filter(user=request.user)  #
Fetch the bots owned by the logged-in user

    if request.method == 'POST':
        if 'delete' in request.POST:  # Check if
there's a request to delete a bot
            bot_id = request.POST.get('delete')  #
Get the ID of the bot to be deleted from the form
            bot = get_object_or_404(Bot, id=bot_id)
# Find the bot by ID, or show 404 if not found
            bot.delete()  # Delete the bot from the
database
            return redirect('manage_bots')  # After
deleting, redirect back to the bot management page

    return render(request, 'bots/manage_bots.html',
{'bots': bots})  # Render the page with the list of
bots owned by the user
```

Code Explanation:

- ***`login_required`*:**
  The `manage_bots` view is protected by the `login_required` decorator, which ensures that only logged-in users can access this page. If someone tries to access the page without logging in, they will be redirected to the login page. This feature is crucial for the security of the application, especially when managing sensitive data like the bots created by the user.
- ***Fetching Bot List:***
  The code `bots = Bot.objects.filter(user=request.user)` is responsible for retrieving all the bots owned by the logged-in user. By filtering based on the `user` attribute, only the bots relevant to that user are displayed on the management page. This is important because we don't want users to see or manage other users' bots.
- ***Deleting Bots:***
  In this part, we handle the logic for deleting bots. If the user presses the delete button for a specific bot on the page, a POST request will be sent to this view. The view then checks if a `delete` parameter exists in the POST request using `if 'delete' in request.POST`. If this parameter is present, it means the user intends to delete a specific bot. The ID of the bot to be deleted is fetched using `request.POST.get('delete')`.

  The function `get_object_or_404` is used to find the bot corresponding to that ID in the database. If the bot is found, it is deleted from the database with `bot.delete()`. If not, the user is redirected to a 404 page, ensuring that only valid bots can be deleted. After the bot is deleted, the user is redirected back to the bot management page using `redirect('manage_bots')`.

- ***Rendering the Template:***
  After successfully retrieving the bot data from the database, the view renders the `manage_bots.html` template using `render(request, 'bots/manage_bots.html', {'bots': bots})`. The data fetched (in this case, the list of bots) is passed to the template as context, allowing the bot list to be displayed to the user.

By using this approach, we ensure that the bot management page securely displays the list of bots and allows users to delete unwanted bots.

In this subsection, we created the `manage_bots` view, which allows users to manage the bots they own. The view loads the list of bots based on the logged-in user and provides secure bot deletion functionality via a POST mechanism. By using `login_required` and `get_object_or_404`, we ensure that only authorized users can access this page and that only valid bots are deleted.

The next step is to create a template to display the bot list, which we will cover in the next subsection. This template will present the bot data in a neat, readable format and provide buttons for deleting bots.

8.4.2 Creating the Template for Bot Management

After setting up the view for managing bots, the next step is to create a template that will display the list of bots owned by the user. This template will serve as an interface for users to view their bot information, such as name, description, token, status, and provide options to edit or delete the bot.

This template is designed to be easy to read and use, utilizing HTML elements such as tables to present data in a structured way. We will also take advantage of Bootstrap features, such as styled buttons and tables, to achieve a more professional look.

Open the `manage_bots.html` file located inside the `apps/templates/bots/` directory. Here, we will write the HTML code to display the list of bots. Below is the code for this template:

```
<!-- File: apps/templates/bots/manage_bots.html -->

{% extends 'dashboard/base.html' %}  <!-- Using the
base template from the dashboard -->

{% block content %}
  <div class="container">
```

```
    <h1>Manage Bots</h1>  <!-- Page title -->

    <!-- Table for displaying the list of bots -->
    <table class="table table-striped">
      <thead>
        <tr>
          <th>ID</th>  <!-- Bot ID column -->
          <th>Name</th>  <!-- Bot name column -->
          <th>Platform</th>  <!-- Bot platform column
-->
          <th>Token</th>  <!-- Bot token column -->
          <th>Status</th>  <!-- Bot status column -->
          <th>Actions</th>  <!-- Column for actions
like edit or delete -->
        </tr>
      </thead>
      <tbody>
        {% for bot in bots %}  <!-- Loop to display
each bot owned by the user -->
          <tr>
            <td>{{ bot.id }}</td>  <!-- Display bot
ID -->
            <td>{{ bot.name }}</td>  <!-- Display bot
name -->
            <td>{{ bot.bot_type }}</td>  <!-- Display
bot type or platform -->
            <td>{{ bot.token_telegram }}</td>  <!--
Display bot token -->
            <td>{{ bot.is_active|
yesno:"Active,Inactive" }}</td>  <!-- Display bot
active or inactive status -->
            <td>
              <a href="" class="btn btn-
primary">Edit</a>  <!-- Button to edit the bot -->

              <!-- Form to delete the bot -->
              <form method="post"
style="display:inline;">
                {% csrf_token %}  <!-- CSRF token for
form security -->
                <button type="submit" name="delete"
value="{{ bot.id }}" onclick="return confirm('Are you
sure you want to delete this bot?');" class="btn btn-
danger">Delete</button>  <!-- Delete button with
confirmation -->
              </form>
            </td>
          </tr>
        {% empty %}  <!-- If no bots are available --
>
          <tr>
            <td colspan="6">No bots available.</td>
<!-- Message displayed when there are no bots -->
          </tr>
        {% endfor %}
      </tbody>
    </table>

    <!-- Button to create a new bot -->
    <a href="{% url 'create_bot' %}" class="btn btn-
primary">Create New Bot</a>
  </div>
{% endblock %}
```

Code Explanation:

- ***`{% for bot in bots %}`:*** This loop is used to display the list of bots owned by the user. Each bot retrieved from the `manage_bots` view will be displayed in the table with relevant columns such as ID, name, platform, token, and status. We also provide options for the user to edit or delete each bot.
- ***Bot List Table:*** A table is used to neatly organize information for each bot. The columns in this table display important details such as the bot ID, bot name, platform type (e.g., Telegram or WhatsApp), the bot's token, and the bot's status (active or inactive). By using Bootstrap classes such as `table` and `table-`

`striped`, the table looks more professional and organized.

- ***Delete Form:*** Each row in the table includes a delete button, represented by a POST form. When the user clicks the delete button, this form sends the ID of the bot to be deleted to the `manage_bots` view. The `csrf_token` feature is included for security, and simple JavaScript is used to display a confirmation message before deleting the bot.
- ***No Bots Message:*** If the user doesn't have any bots, the `{% empty %}` section will be executed, displaying the message "No bots available." This ensures that the page remains informative even if the bot list is empty.
- ***Create New Bot Link:*** At the bottom of the page, there is a link that allows users to create a new bot. This link directs users to the `create_bot.html` page, where they can create a new bot with their desired platform and information.

The `manage_bots.html` template provides a clear and easy-to-use interface for users to manage their bots. With a structured table, users can view detailed information about their bots and take actions such as editing or deleting them with ease. Additionally, security features like the CSRF token and JavaScript confirmation ensure that bot deletion is done safely.

The next step is to link this view and template with URL routing so that this page can be accessed through the browser. The next subchapter will cover how we add a URL for the bot management page in our Django application.

8.4.3 Setting URL Routing for Managing Bots

Finally, we need to set up routing so users can access the bot management page. We will add a route for this page in the `apps/bots/urls.py` file.

Open the `apps/bots/urls.py` file and add the following route:

```
# File: apps/bots/urls.py
from django.urls import path
from . import views

urlpatterns = [
    # URL for the bot creation page
    path('manage/', views.manage_bots,
name='manage_bots'),  # URL for the bot management
page
]
```

Code Explanation:

- **`path('manage/', views.manage_bots, name='manage_bots')`**: This URL connects the bot management page with the `manage_bots` view. Users can access this page via `/manage/`.

With this routing, users can access the bot management page by typing `/manage/` in their browser.

At this point, we have successfully created the bot management page. Users can view the list of bots they have created and delete any unnecessary ones. With the view, template, and URL routing set up, we now have a basic page for managing bots.

8.5 Complete Code

Let's review all the code we've created to ensure there are no mistakes. Below is the complete code we've written so far:

8.5.1 Models.py

```
# File: apps/bots/models.py

from django.db import models
from django.contrib.auth.models import User

class Bot(models.Model):
    BOT_CHOICES = [
        ('telegram', 'Telegram'),
        ('whatsapp', 'WhatsApp'),
    ]
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # The bot will be linked
to the user
    bot_type = models.CharField(max_length=20,
choices=BOT_CHOICES)  # Bot type: Telegram or
WhatsApp

    token_telegram = models.CharField(max_length=255,
unique=True, help_text="Enter a unique bot token.")
# Telegram token
    webhook_url = models.URLField(max_length=255,
blank=True, help_text="The bot webhook URL will be
set automatically.")  # Webhook URL
```

```
    is_active = models.BooleanField(default=True,
help_text="Bot active status.")  # Whether the bot is
active or not

    # Automatic messages
    start_message = models.TextField(blank=True,
default="Welcome to the bot!", help_text="Message
sent when a user types /start.")
    help_message = models.TextField(blank=True,
default="Here’s how to use the bot.",
help_text="Message sent when a user types /help.")

    messages = models.JSONField(default=list,
blank=True, help_text="List of received messages and
their responses in JSON format.")  # JSON format

    # Timestamps
    created_at =
models.DateTimeField(auto_now_add=True)  # Date the
bot was created
    updated_at = models.DateTimeField(auto_now=True)
# Date the bot was last updated

    name = models.CharField(max_length=255,
blank=True, default="My Bot", help_text="Bot name.")
# Bot name
    description = models.TextField(blank=True,
default="This is my first bot", help_text="Bot
description.")  # Bot description

    def __str__(self):
        return f'{self.bot_type.capitalize()} Bot -
{self.user.username}'

    class Meta:
        verbose_name = "Bot"
        verbose_name_plural = "Bots"
```

8.5.2 Forms.py

```
# File: apps/bots/forms.py

from django import forms
from .models import Bot  # Importing the Bot model

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot  # Linking the form to the Bot
model
        fields = ['bot_type', 'token_telegram',
'name', 'description', 'is_active', 'start_message',
'help_message']  # Fields to be displayed in the form
        widgets = {
            'name': forms.TextInput(attrs={'class':
'form-control', 'placeholder': 'Enter bot name'}),
            'description':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Bot description'}),
        }
        help_texts = {
            'token_telegram': 'Enter the unique token
received from the Telegram platform.',
            'name': 'You can change the bot name as
you wish.',
            'description': 'A brief description of
the bot and its functions.',
        }
```

8.5.3 Views.py

```
# File: apps/bots/views.py

from django.shortcuts import render, redirect,
get_object_or_404
from django.contrib.auth.decorators import
login_required
```

```
from .forms import BotForm
from .models import Bot

@login_required
def create_bot(request):
    """
    View to create a new bot. Not yet connected to
webhook and activity log.
    """
    if request.method == 'POST':
        form = BotForm(request.POST)
        if form.is_valid():
            bot = form.save(commit=False)
            bot.user = request.user  # Saving the bot
with user information
            bot.save()  # Save bot to the database

            # Temporarily, there's no integration
with webhook or activity log.
            # Webhook and logging will be integrated
in the next chapter.

            return redirect('manage_bots')  # After
success, redirect to bot management page
    else:
        form = BotForm()

    return render(request, 'bots/create_bot.html',
{'form': form})

@login_required  # Ensure only logged-in users can
access this view
def manage_bots(request):
    bots = Bot.objects.filter(user=request.user)  #
Retrieve list of bots owned by the logged-in user

    if request.method == 'POST':
        if 'delete' in request.POST:  # Check if a
request to delete a bot was made
            bot_id = request.POST.get('delete')  #
Get the ID of the bot to delete from the form
```

```
            bot = get_object_or_404(Bot, id=bot_id)
# Find the bot by ID, or show 404 if not found
            bot.delete()  # Delete the bot from the
database
            return redirect('manage_bots')  # After
bot deletion, return to the bot management page

    return render(request, 'bots/manage_bots.html',
{'bots': bots})  # Render the page with the list of
bots owned by the user
```

8.5.4 URLS.py

```
# File: apps/bots/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('create/', views.create_bot,
name='create_bot'),  # URL for the bot creation page
    path('manage/', views.manage_bots,
name='manage_bots'),  # URL for the bot management
page
]
```

With all this code, it is expected that there will be no errors in the application, ensuring a smooth and functional user experience.

Chapter Conclusion

This chapter has outlined the essential steps in setting up the bot management page within a Django application. We began by setting up the model and form for the bot, where the model defines the data structure of the bot, and the form allows users to easily input data. Next, we discussed the creation of pages to create, edit, and manage bots, which involved developing views, templates, and configuring appropriate URL routing.

Each section explained how to connect the frontend and backend, as well as how to integrate bots with webhooks to ensure smooth communication. At the end of the chapter, all the discussed code was gathered in one place to facilitate an overall understanding. By following the guide in this chapter, we can build a well-structured and functional bot management page within a Django application.

With all these components, we have established a solid foundation for the bot management page. Users can now view and manage their bots in an intuitive way.

The next steps may include adding additional features, such as displaying more information about the bots or expanding the functionality of the page to include other features relevant to user needs.

Overall, this section has provided a comprehensive guide on how to create and manage a bot management page in a Django application. With this foundation, you are now ready to further develop and add additional features based on the needs of your project.

Chapter 9 - Token Validation for Bots

In this section, we will delve into the process of token validation used for bots and how to implement it effectively within an application. Token validation is a crucial step to ensure that only valid tokens, which comply with the specified format, can be used to create and manage bots. Proper validation prevents errors and maintains system integrity, allowing the bots to function optimally and securely. Let’s begin by understanding what a token is, the importance of validation, and how we can implement it in our application.

9.1 Introduction

Building an effective bot in a web application requires a deep understanding of the tokens used for authentication and identification. A token is a unique string that acts as an access key to connect a bot to a specific platform, such as Telegram, Discord, or other platforms. In this context, token validation becomes an important step to ensure that only valid and correctly formatted tokens are accepted when creating a bot.

Token validation not only protects the system from invalid input but also enhances application security by ensuring that only users with the correct token can create and manage bots. If the token provided by the user is incorrect or invalid, the bot creation process must be rejected, and a clear error message should be communicated to the user.

One of the main challenges in token validation is the variety of token formats used by different platforms. For example, a Telegram token has a specific format that differs from tokens from other platforms. Therefore, it is important for us to understand the format of each token and how to implement appropriate validation in our Django application.

In the following subsections, we will explore common token formats and how to implement token validation in a Django application. We will also discuss creating models for tokens and handling errors that may occur when users attempt to enter invalid tokens. Through this discussion, we aim to ensure a smooth and secure bot creation process so that users can focus on bot functionality without worrying about errors caused by invalid input.

With a solid understanding of token validation, we can enhance user experience and ensure that our application functions well under various conditions. Next, let's discuss the common token formats to help us understand what needs to be validated when users enter a token for their bot.

9.1.1 The Importance of Token Validation

In developing platforms that allow users to create bots, one of the most important elements to consider is token validation. Tokens serve as authentication keys that enable bots to communicate with the API of the platform being used, such as Telegram or WhatsApp. Without proper validation, the system is vulnerable to input errors, which can lead to operational failures or even jeopardize the security of the entire platform.

Token validation is a critical step in bot development, especially when interacting with multiple platforms. Tokens act as a bridge connecting our bots to service platforms, and errors in this process can have serious consequences, ranging from data leaks to unauthorized access. Therefore, it is essential to ensure that the tokens provided by users meet specific criteria and are technically valid.

A token can be considered as the "identity" that a bot holds to access external services. For example, on the Telegram platform, each bot is identified by a unique token issued by BotFather. This token must follow a very specific format, combining the bot’s ID and secret key, separated by a colon. Such tokens ensure that the bot connected to the API is legitimate.

However, problems arise when users enter the token incorrectly or attempt to input an invalid value. For example, if users enter random text or a string that does not follow the expected format, the Telegram API will not recognize the bot's request. As a result, the connection between the bot and the API fails, and the bot will not function as intended. The same applies to other platforms such as WhatsApp via Twilio, where parameters like `account_sid`, `auth_token`, and `whatsapp_number` must adhere to specified formats.

Moreover, token validation is not only about ensuring the correct format but also about maintaining security. An incorrect or arbitrary token can pose a security risk to the system. For instance, without validation, users could enter tokens that might potentially exploit or access unauthorized data. This opens the door to various threats, including API misuse or even denial of service (DoS) attacks that could slow down or halt the entire system.

This is where the importance of token validation becomes critical. In our system, token validation is carried out both on the server and client sides. This means that before a token is stored in the database, the system must ensure that it meets the requirements of the associated platform, whether it be Telegram or WhatsApp. Additionally, the system should provide clear feedback to users if the token they entered is invalid.

For example, if a user attempts to enter a Telegram token in an incorrect format like "this_is_a_wrong_token", the system will display an informative error message such as "The Telegram token you entered is invalid. Make sure the token is formatted like this: 5898177748
." This way, users can quickly correct their mistakes without confusion.

In Django, token validation can be applied both at the model and form levels. Validation at the model level ensures that the data entered into the database is in the correct format, while validation at the form level helps provide faster feedback to users when they are filling out forms. This validation can be done by adding custom validation in `models.py` or `forms.py`, depending on the approach we choose.

For instance, if we choose to validate at the form level, we can define a `clean` method in `forms.py` to check whether the token entered by the user follows the required format. For example, for Telegram tokens, we can verify that the token consists of two parts separated by a colon and that the second part has the correct length for a secret key.

```python
# File: forms.py
from django import forms
import re

class TelegramTokenForm(forms.Form):
    token = forms.CharField(max_length=255,
required=True)

    def clean_token(self):
        token = self.cleaned_data.get('token')
        # Check if the token has the correct format
(ID:Secret Key)
        if not re.match(r'^\d{9}:[A-Za-z0-9_-]{35}$',
token):
            raise forms.ValidationError("The Telegram
token is invalid. Ensure the token is formatted like
this:
5898177748:AAFBMw_gSELJWsFvvvOFueRGQ459cHSaNCk")
        return token
```

In the example above, we use a regular expression to ensure that the token entered by the user follows the correct pattern, which consists of a nine-digit numeric ID, followed by a colon, and then a 35-character alphanumeric secret key. If the token does not match this pattern, Django will raise an error message, providing clear guidance to the user.

This validation approach not only improves user experience but also provides an additional layer of protection against potential errors or security threats arising from invalid input. A robust validation system is crucial in maintaining platform reliability and security, ensuring that the bots created by users can operate smoothly and securely.

By implementing thorough and effective validation, we ensure that only valid tokens can be used to operate bots on the platform. This is an essential first step in building a reliable and secure platform.

9.1.2 Types of Tokens and Their Uses

In the bot development ecosystem, each platform has its own authentication mechanism, typically in the form of tokens. These tokens serve as digital keys that allow bots to communicate with the platform's servers, such as Telegram or WhatsApp. Each platform requires tokens with different formats and rules, which developers must understand to ensure successful integration.

In the context of bot development, there are various types of tokens used, each with specific functionalities and purposes.

Understanding these token types is crucial because it helps in the proper validation and usage within our application.

One of the most common token types is the ***API Token***. This token is typically used for authentication in communication between an application and a server. Whenever our bot makes an API request, such as fetching data or sending a message, the API Token is included in the request to ensure that it originates from an authorized source. For example, the Telegram API token is used to authenticate our bot when communicating with Telegram servers.

These tokens are usually long strings containing a combination of letters and numbers. It’s essential to ensure that these tokens are difficult to guess and kept confidential. To validate an API token, we need to check its format and length according to the platform's specifications.

In addition to API Tokens, there are also ***OAuth Tokens***, which are often used in systems that require user authorization. OAuth is a protocol that allows users to grant access to third-party applications without sharing their login information. In the context of bots, once the user grants permission, our application receives an OAuth Token that can be used to access the user's data. This process is more secure and reduces the risk of personal information leakage.

The use of OAuth Tokens requires an understanding of the authorization flow. For example, in a web application scenario, we need to direct users to the platform’s login page to obtain permission and then receive the necessary token. In this case, token validation is also crucial, as we must ensure that the token is still valid and has the appropriate permissions to access data.

There is also the ***Bearer Token***, a type of token commonly used in token-based authentication. Bearer Tokens are used in the HTTP request header, allowing the server to identify users without requiring additional authentication. When using Bearer Tokens, it's important to validate the token in every request to ensure the access granted remains valid.

Moreover, some platforms also use ***Refresh Tokens***, which allow users to obtain a new token without having to log in again. Refresh Tokens typically have a longer lifespan than regular tokens. In this case, validating the Refresh Token is important to ensure that it hasn't expired and can still be used to gain new access.

Telegram Tokens

On the Telegram platform, every bot is identified and authorized to interact with the Telegram API using a unique token. This token is provided by BotFather, a special Telegram bot that manages other bots. When users create a new bot through BotFather, they will receive a token that looks like this:

```
5898177748:AAFBMw_gSELJWsFvvvOFueRGQ459cHSa
NCk
```

This token consists of two main parts separated by a colon (`:`). The first part is the bot ID, which is a series of numbers that uniquely identifies the bot in the Telegram system. The second part is the secret key that allows the bot to perform operations like sending messages or receiving updates from the Telegram API.

The Telegram token is used in every API request made by the bot. When the bot attempts to send a message or retrieve information from Telegram, this token is included in the request header to ensure that the request is legitimate and comes from an authorized bot. For example, if we want to send a message via the bot, we need to use this token to access the Telegram API endpoint.

Telegram’s validation system is strict, as the bot will only accept and execute requests that include a valid token. If the token is incorrect or does not match the expected format, the request will be rejected, and the bot will not be able to operate properly.

WhatsApp Tokens via Twilio

On the other hand, WhatsApp does not have direct authentication tokens like Telegram. To integrate a WhatsApp bot, we need to use third-party services such as Twilio, which acts as an intermediary between the bot and the WhatsApp API. On Twilio, authentication is done using three main parameters: `account_sid`, `auth_token`, and a verified WhatsApp number.

- ***Account SID*** is a unique identifier provided by Twilio for each registered account. It works similarly to a user ID, uniquely identifying the source of the API request.
- ***Auth Token*** is a secret key provided by Twilio for authorization. Like the secret key in Telegram, this `auth_token` must be kept confidential as all API requests made by the bot will require this token for verification.
- ***WhatsApp Number*** is the phone number registered with Twilio and authorized to send and receive WhatsApp messages through the Twilio API. This number must be in international format, for example:

```
+14155238886
```

These three parameters (`account_sid`, `auth_token`, and WhatsApp number) are used together to authenticate every request made by the bot to the WhatsApp API. When the bot attempts to send a message via WhatsApp, Twilio will verify whether the request comes from a valid and correctly authenticated account using a combination of `account_sid` and `auth_token`.

Token Usage in Authentication

Both on Telegram and WhatsApp via Twilio, tokens play a crucial role in maintaining secure communication between the bot and the API. Every request sent by the bot must include a valid authentication token or parameter. If the token is invalid or has expired, the request will be rejected by the API server.

For Telegram, authentication is entirely done through the token generated by BotFather. Every time the bot sends an API request, such as sending a message or receiving an update, this token must be included in every request. Telegram will verify whether the token is valid before processing the request.

For WhatsApp via Twilio, authentication is done by verifying a combination of `account_sid`, `auth_token`, and WhatsApp number. These three parameters must match those registered in the Twilio account, and all API requests submitted by the bot will be verified using this data.

It’s important to note that keeping tokens secure is essential. If these tokens fall into the wrong hands, unauthorized parties could misuse them to access the API or even operate the bot on behalf of the user. Therefore, in bot platform development, we must ensure that tokens are always stored securely and strong validation is applied to verify the authenticity of these tokens.

In a Django implementation, we can ensure this validation occurs both at the model and form level to guarantee that tokens entered by users have the correct format and meet the API requirements of the platform in use. For example, by adding validation logic in `models.py` or `forms.py`, we can ensure that the token received is valid before being saved to the database or used to process API requests.

This validation not only ensures that the bot functions properly but also protects the system from unauthorized token misuse.

Understanding these various types of tokens will help us design more effective and secure validation systems. Each type of token has its own characteristics and rules that must be followed. By having a deep understanding of how and when each token is used, we can build bot applications that are not only functional but also secure and reliable. In the next sub-chapter, we will discuss how to implement token validation in our Django code, ensuring that the bots created by users meet all the specified requirements.

9.2 Understanding Token Format

Tokens are crucial elements in bot authentication systems, especially on platforms like Telegram and WhatsApp. Each token has a different format based on the needs and standards of the platform. Understanding token format is a vital first step in ensuring correct token validation. In this section, we will review the token format used by Telegram in detail. A deep understanding of this token structure will help us validate it within Django applications, ensuring that every token submitted by users complies with the platform's standards.

9.2.1 Telegram Token Format

In developing a Telegram bot, the token format is critical to ensuring the bot functions properly. The Telegram token is one of the primary elements that links the bot to the Telegram API. This token is provided by BotFather when a developer creates a new bot. Each token has a unique format and consists of two main parts: `bot_id` and `secret_key`, which are separated by a colon (:).

For example, a typical Telegram token looks like this:

```
5898177748:AAFBMw_gSELJWsFvvvOFueRGQ459cHSaNCk
```

Token Structure Explanation:

1. ***First part (bot_id)***:
   This is a series of numbers representing the bot's ID in the Telegram system. This ID is a unique identifier assigned to each bot created by the user. In the example token above, the `bot_id` is `5898177748`. Each bot created through BotFather receives a different `bot_id`. This bot_id is important as it serves as one of the keys in identifying the bot by Telegram.
2. ***Second part (secret_key)***:
   This is a secret key used to secure the bot. The `secret_key` is provided by Telegram and must be kept confidential, as it allows the bot to perform operations through the Telegram API. The `secret_key` typically consists of uppercase letters, lowercase letters, numbers, and some special characters like underscores (_). In the example above, the `secret_key` is `AAFBMw_gSELJWsFvvvOFueRGQ459cHSaNCk`.

Why Is This Format Important?

Telegram uses both parts of the token to ensure that only properly authorized bots can access their API. When a bot sends a request to the Telegram API, this token is used to verify the bot's identity. Telegram then validates whether the `bot_id` and `secret_key` are correct and valid. If the token is incorrect, the API will return an error, and the bot will not function properly.

It is crucial to ensure that this token is securely stored and not shared with others. If this token falls into the wrong hands, the person could control the bot and perform unwanted actions. Therefore, we need to implement validation steps to ensure that the token provided by users matches the expected format.

To validate the format of a Telegram token, we can use regular expressions. Regular expressions allow us to check whether the entered token follows the appropriate pattern. For example, we can write a validation function in our Django application to check the token format as follows:

```
import re

def validate_telegram_token(token):
    # Check if the token matches the Telegram format
    token_pattern = r'^\d{9}:[A-Za-z0-9_-]{35}$'  #
Format for bot ID and token
    return re.match(token_pattern, token) is not None
# Return True if it matches
```

In the code above, we use the `re` module to check whether the entered token matches the specified pattern. If the token is valid, this function will return `True`, indicating that the token follows the expected format. If not, we return `False`, meaning the token is invalid.

For example, in a Django application, we can also add validation logic at the form level to ensure the token has the correct format.

This can be done by checking whether the token consists of two parts separated by a colon (:), and whether both parts meet the expected length and format criteria.

```
# File: forms.py

from django import forms
from django.core.exceptions import ValidationError

class TelegramBotForm(forms.Form):
    token = forms.CharField(max_length=100,
help_text="Enter your Telegram bot token")

    def clean_token(self):
        token = self.cleaned_data.get('token')
        if ':' not in token:
            raise ValidationError("The token must
consist of two parts separated by a colon (:)")

        bot_id, secret_key = token.split(':', 1)
        if not bot_id.isdigit():
            raise ValidationError("The first part of
the token must be numeric (bot_id)")
        if len(secret_key) < 35:  # The length of
secret_key can vary
            raise ValidationError("The secret key
(secret_key) must be sufficiently long and valid")

        return token
```

In the above code, we ensure that the token has the correct format by checking the presence of a colon (:), validating that `bot_id` is a number, and ensuring the `secret_key` is long enough to be a valid key. If the token does not meet these criteria, the system will return a clear error message to the user.

Validation at the Model Level

In addition to form-level validation, we can also implement validation at the model level to ensure that every token stored in the database is valid. Here is an example of how we can add validation to the model:

```python
# File: models.py

from django.db import models
from django.core.exceptions import ValidationError

def validate_telegram_token(token):
    if ':' not in token:
        raise ValidationError("The token must consist
of two parts separated by a colon (:)")

    bot_id, secret_key = token.split(':', 1)
    if not bot_id.isdigit():
        raise ValidationError("The first part of the
token must be numeric (bot_id)")
    if len(secret_key) < 35:
        raise ValidationError("The secret key
(secret_key) must be sufficiently long and valid")

class TelegramBot(models.Model):
    name = models.CharField(max_length=100)
    token = models.CharField(max_length=100,
validators=[validate_telegram_token])

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

By using model-level validation as shown above, we ensure that Telegram tokens always follow the correct format whenever they are saved in the database. If the token entered is not valid, the data will not be saved, and the user will receive an appropriate error message.

Understanding the Telegram token format is not only essential for validation but also for understanding how Telegram authenticates bots on its platform. In the next section, we will discuss token formats on other platforms, such as WhatsApp, which have a different but equally important authentication approach.

By understanding the Telegram token format and applying the proper validation steps, we can ensure that the bots created by users have legitimate access to the Telegram API. Next, we will discuss how to implement this validation within our Django application and display informative error messages when users input an invalid token.

9.2.2 WhatsApp Token Format

WhatsApp has a different approach when it comes to bot authentication, where its integration is done through Twilio, a third-party service that enables messaging via API. In this integration, WhatsApp bot authentication does not rely on a single token like Telegram but involves several critical parameters that must be validated simultaneously. For each WhatsApp bot integrated with Twilio, the three key elements to consider are the `account_sid`, `auth_token`, and `whatsapp_number`. These three elements must have the correct and valid format for the bot communication via Twilio to function smoothly.

account_sid

`account_sid` is a unique identifier provided by Twilio to each account. The format of `account_sid` always begins with the prefix "AC", followed by a unique combination of letters and numbers, as shown in the example below:

```
ACa7914f90b10b1509ab0780f009d4a9fe
```

The first part of the SID (AC) indicates that the SID is related to the Twilio account. The rest of the string is a combination of characters that serve as a unique identity for that account. In a Django implementation, we must ensure that the `account_sid` format starts with "AC" and has a specific length that meets Twilio's standard.

auth_token

In addition to the `account_sid`, Twilio also provides an `auth_token` as a secret key used to authenticate API requests. This token consists of a random combination of letters and numbers and must be kept confidential as it grants full access to Twilio services. An example of a valid `auth_token` is:

```
07dca14a66ddc70305c5f47588ab84f2
```

To ensure security, the `auth_token` must be validated to meet a minimum length and contain the expected characters. If a user inputs an invalid `auth_token`, the application should provide an informative error message while still protecting sensitive information.

whatsapp_number

The last parameter used is the `whatsapp_number`, which is the phone number registered with Twilio and used to send and receive messages through WhatsApp. The format of this WhatsApp number must follow the international standard, beginning with a plus sign (+) followed by the country code and phone number. Example:

```
+14155238886
```

In this example, +1 is the country code for the United States, and 4155238886 is the phone number used. For validation, the WhatsApp number must be checked to ensure it conforms to the international phone number standard. Failure to validate this phone number could result in undelivered messages or failure of the bot to communicate with WhatsApp users.

9.2.3 Why is This Validation Important?

Validating these three parameters is crucial to ensuring that the bot can operate smoothly via Twilio. An error in any of the parameters, such as the `account_sid` or `auth_token`, could lead to authentication failures with Twilio's API, preventing the bot from sending or receiving messages. Furthermore, if the `whatsapp_number` is not correctly validated, messages may not reach the intended users.

In our Django application, we can leverage validation at the form or model level to ensure these three parameters are always in the correct format before the data is saved or used.

Here is an example of how validation for these three elements can be implemented in a Django form:

```
# File: forms.py

from django import forms
from django.core.exceptions import ValidationError
import re

class WhatsAppBotForm(forms.Form):
    account_sid = forms.CharField(max_length=34,
help_text="Enter the Twilio Account SID")
    auth_token = forms.CharField(max_length=32,
help_text="Enter the Twilio Auth Token")
    whatsapp_number = forms.CharField(max_length=15,
help_text="Enter the registered WhatsApp number")

    def clean_account_sid(self):
        account_sid =
self.cleaned_data.get('account_sid')
        if not account_sid.startswith('AC'):
            raise ValidationError("Account SID must
start with 'AC'")
        if len(account_sid) != 34:
            raise ValidationError("Account SID must
be 34 characters long")
        return account_sid

    def clean_auth_token(self):
        auth_token =
self.cleaned_data.get('auth_token')
        if len(auth_token) < 32:
            raise ValidationError("Auth Token must be
32 characters long")
        return auth_token

    def clean_whatsapp_number(self):
        whatsapp_number =
self.cleaned_data.get('whatsapp_number')
        if not re.match(r'^\+?\d{10,15}$',
whatsapp_number):
```

```
            raise ValidationError("WhatsApp number
must be in a valid international format")
        return whatsapp_number
```

In the code above, we perform several checks:

1. `account_sid` must start with "AC" and be 34 characters long.
2. `auth_token` must have a minimum length of 32 characters.
3. `whatsapp_number` must be in an international format, starting with a plus sign (+) and containing 10 to 15 digits.

If any of these parameters do not meet the requirements, the system will display an error message to the user, providing clear guidance on what needs to be corrected.

In addition to form validation, these validations can also be applied at the model level to ensure data integrity when stored. An example of model implementation might look like this:

```
# File: models.py

from django.db import models
from django.core.exceptions import ValidationError
import re

def validate_whatsapp_number(number):
    if not re.match(r'^\+?\d{10,15}$', number):
        raise ValidationError("WhatsApp number must
be in international format")

class WhatsAppBot(models.Model):
    account_sid = models.CharField(max_length=34)
    auth_token = models.CharField(max_length=32)
    whatsapp_number = models.CharField(max_length=15,
validators=[validate_whatsapp_number])

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

Model-level validation adds an extra layer of protection, ensuring that any data stored in the database is in the correct format.

With proper validation, we can prevent errors from the start and ensure that the WhatsApp bot communicates correctly via Twilio. This validation is a crucial step in ensuring that the authentication system works smoothly, providing users with a better experience when configuring their bots.

9.2.4 Token Format for Other Platforms

As bot platforms evolve beyond Telegram and WhatsApp, it is highly likely that we will encounter different types of tokens depending on the platform specifications. Therefore, designing a flexible token validation system is a crucial step to ensure that our application can adapt and grow easily to accommodate various token types from new platforms in the future. This approach allows our application not only to support existing platforms but also to be ready for platforms we have not yet integrated.

Besides Telegram, many other platforms provide APIs that allow developers to build integrated bots or applications. Each platform typically has a unique token format, and understanding these formats is essential to ensure that our application functions properly.

Flexible Approach to Token Validation

To support various platforms in the future, we need to ensure that the validation system does not rely on a single token format. If we create a validation system that only fits Telegram or WhatsApp, adding a new platform will require significant code changes. Instead, we can design a dynamic system where the token format can be easily modified or added without altering the entire codebase.

Using Flexible Regex

One way to achieve this flexibility is by using Regular Expressions (regex) that can be adjusted as needed. For example, each platform may have a specific token pattern, and the regex used to validate those tokens can be stored dynamically or determined based on the platform selected by the user.

In Django, this approach can be implemented at the form or model level. Here's an example of how dynamic validation for various tokens can be implemented in a Django form:

```
# File: forms.py

from django import forms
from django.core.exceptions import ValidationError
import re

class BotTokenForm(forms.Form):
    platform =
forms.ChoiceField(choices=[('telegram', 'Telegram'),
('whatsapp', 'WhatsApp'), ('other_platform', 'Other
Platform')])
    token = forms.CharField(max_length=255,
help_text="Enter the token according to the selected
platform")

    def clean_token(self):
        token = self.cleaned_data.get('token')
        platform = self.cleaned_data.get('platform')

        # Validation based on platform
        if platform == 'telegram':
            if not re.match(r'^\d{10}:[A-Za-z0-9_-]
{35}$', token):
                raise ValidationError("Invalid
Telegram token")
        elif platform == 'whatsapp':
            if not re.match(r'^[A-Za-z0-9]{32}$',
token):  # Twilio Auth Token for example
                raise ValidationError("Invalid
WhatsApp token")
        else:
            # Default or other platform token
validation
            if not re.match(r'^[A-Za-z0-9_-]
{20,255}$', token):  # Common example for other
platform tokens
                raise ValidationError("Invalid token
for this platform")

        return token
```

In the code above, we see how a Django form is set up to accept token input based on the selected platform. When a user selects a platform, the system adjusts the token validation according to the pattern for that platform. For example, a Telegram token is validated against the pattern `^\d{10}:[A-Za-z0-9_-]`

`{35}$`, while a WhatsApp token might have a length of 32 alphanumeric characters.

For platforms that are not yet integrated, we can use a more general regex pattern, such as `^[A-Za-z0-9_-]{20,255}$`, which accepts tokens between 20 and 255 characters long. This is a sufficiently flexible pattern to handle various token types that might be used by future bot platforms.

Easily Accommodating New Platforms

With the above approach, when a new platform is introduced, we only need to add additional validation logic in the relevant section without having to change the entire code structure. For example, if we add support for platforms like Slack or Discord in the future, we can add new validation patterns based on their token formats without affecting existing validations for Telegram or WhatsApp.

Moreover, the system can be made even more flexible by using a database or external configuration to store token validation patterns for each platform. This allows system administrators to add or change token formats without modifying the main application code.

For example, here’s how we can use a Django model to store dynamic token patterns:

```
# File: models.py
```

```
from django.db import models
from django.core.exceptions import ValidationError
import re

class Platform(models.Model):
    name = models.CharField(max_length=50)
    token_pattern = models.CharField(max_length=255)
# Store the regex pattern for the token

    def __str__(self):
        return self.name

    def validate_token(self, token):
        if not re.match(self.token_pattern, token):
            raise ValidationError(f"Invalid token for
platform {self.name}")
```

In the model above, each platform can have its own regex token pattern stored in the database. When a new platform is added, the administrator simply adds a new entry to the `Platform` model along with its token pattern. Then, when the form validates a token, it can retrieve the pattern from the database and use it for validation.

```
# File: forms.py

class BotTokenForm(forms.Form):
    platform =
forms.ModelChoiceField(queryset=Platform.objects.all(
))
    token = forms.CharField(max_length=255,
help_text="Enter the token according to the selected
platform")

    def clean_token(self):
        token = self.cleaned_data.get('token')
        platform = self.cleaned_data.get('platform')

        # Validate token based on the selected
platform
```

```
        platform.validate_token(token)

        return token
```

With this approach, we not only create a flexible system but also one that is easily adaptable. Whenever a new platform appears, we simply update the database without touching the main application code.

Each platform typically has its own authentication mechanism, whether it's a simple token like Telegram or a more complex combination like WhatsApp through Twilio. Other bot platforms like Facebook Messenger, Slack, or Discord also have their own API tokens. Therefore, flexibility in token validation is crucial for maintaining system compatibility.

For instance, let's look at the common token format used by platforms like Discord and Slack. A Discord token usually has a different structure from a Telegram token. Discord tokens consist of a long alphanumeric string generated by the Discord system, and may look like this:

`ODI0MTg2ODI3NzQ2NDAwMjM5.YDtrEw.3f5rU8Eqz4Mb8zQW_p-QZPQR7Bk`. This token provides full access to a bot created within a Discord server.

To validate a Discord token, we can use a similar approach as we did for Telegram tokens, but with a pattern specific to Discord. We can use a regular expression to check if the token entered by the user is valid. Here is an example of a validation function for Discord tokens:

```
import re

def validate_discord_token(token):
    # Check if the token matches the Discord format
    token_pattern = r'^[A-Za-z0-9]{24}\.[A-Za-z0-9]
{6}\.[A-Za-z0-9_]{27}$'  # Discord token format
    return re.match(token_pattern, token) is not None
# Returns True if the token is valid
```

In the code above, we define a pattern for Discord tokens and use the `re.match` function to check whether the token provided by the user matches the expected format. If the token is valid, this function will return `True`, indicating that the token can be used to access the Discord API.

Similarly, for Slack, the token typically starts with `xoxb-` followed by a series of numbers and letters. For example, a Slack token might look like this: `xoxb-123456789012-1234567890123-ABCdefGhIJKlmnoPQRstuVWXyz`. Token validation for Slack can also be done using an appropriate regular expression.

By understanding and applying proper validation for tokens from various platforms, we can ensure that the bot or application we build has legitimate access and is not prone to misusage. In the next subsection, we will discuss practical implementation of token validation in our Django application, as well as how to display clear and informative error messages when users enter invalid tokens.

9.2.5 Why is Token Format Important?

Understanding token format is crucial in application development, especially when working with various platforms that provide APIs for integration. A token is not merely a random string of characters; it has a specific structure that dictates how it should be used and validated.

First, token format helps prevent errors. When users are required to enter a token, they may mistype or include invalid characters. By validating the token format, we can give immediate feedback to the user if the token they input is incorrect. This not only enhances user experience but also prevents potential issues in communication with the API.

Second, understanding token format helps maintain application security. An invalid token can be a gateway to potential misuse. If we don’t validate tokens properly, unauthorized users may attempt to access the API and perform unauthorized actions. By applying strict validation based on a defined format, we can add an essential layer of security to our application.

Third, knowing the token format allows developers to debug more easily. When issues arise in communication between the application and the API, understanding the correct token format can help determine if the problem stems from an invalid token. For example, if the API returns an authentication error, developers can quickly check the token used to ensure its format is correct and there were no input errors from the user.

In our Django project, if we allow users to input tokens without strict validation, we might face bigger issues when trying to connect the application to the API. Therefore, implementing proper token format validation is not just a good practice; it is an essential step in developing secure and efficient software.

With this understanding, we can proceed to the next step in the token validation process, which involves practical implementation within our Django application. In the following section, we will explore how to effectively implement this token validation to ensure that all tokens entered by users meet the required criteria.

9.3 Implementing Token Validation In The Project

In developing a Django application that creates and manages bots, one crucial step is to ensure that the tokens entered by users are valid and unique. In this section, we will discuss how to define a model for tokens in the context of the bot we previously created.

9.3.1 Implementing Validation in the Model

In a bot application, it is important to ensure that all data entered into the model is properly validated before being saved to the database. By adding validation, we can prevent data errors such as invalid tokens or bot names that are too short. This validation also improves the user experience by providing more informative error messages when an issue occurs.

In the previously defined ***Bot*** model, we have included an attribute for the token. The token attribute is declared as a ***CharField*** with a maximum length of 255 characters, and we have added the `unique=True` option to ensure that each token entered is unique. This is important because we don't want two bots to have the same token, which could cause conflicts when communicating with the relevant API.

Here is the relevant model code snippet:

```
# File: apps/bots/models.py
class Bot(models.Model):
    # ... other code ...
    token = models.CharField(max_length=255,
unique=True, help_text="Enter a unique bot token.",
blank=False, null=False)
    # ... other code ...
```

With this setup, whenever a user tries to save a new bot, Django will automatically check if the token already exists in the database. If it does, Django will return an error and prevent invalid data from being saved.

9.3.2 Associating the Token with the Bot

Associating the token with the bot also means considering how the token will be used when interacting with APIs. For instance, if we use a Telegram or WhatsApp token, we must ensure that the entered token matches the correct format for each platform.

To add token validation in the Bot model, we need to adjust the model so it can validate Telegram and WhatsApp tokens before storing them in the database. This validation can be done using the `clean()` method in the Django model. This method will automatically be called when the model is saved, and we can add validation logic within it.

Here is the modified code to add validation in the Bot model:

```
# File: apps/bots/models.py

from django.db import models
from django.contrib.auth.models import User
from django.core.exceptions import ValidationError  #
Import for validation
import re  # Import to validate token patterns

class Bot(models.Model):
    BOT_CHOICES = [
        ('telegram', 'Telegram'),
        ('whatsapp', 'WhatsApp'),
    ]
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # Bot associated with user
    bot_type = models.CharField(max_length=20,
choices=BOT_CHOICES)  # Bot type: Telegram or
WhatsApp

    token_telegram = models.CharField(max_length=255,
unique=True, help_text="Enter a unique bot token.")
# Telegram token
    webhook_url = models.URLField(max_length=255,
blank=True, help_text="Bot webhook URL will be
automatically set.")  # Webhook URL

    is_active = models.BooleanField(default=True,
help_text="Is the bot active or not?")  # Bot active
status

    # Automatic messages
    start_message = models.TextField(blank=True,
default="Welcome to the bot!", help_text="Message
sent when user types /start.")
```

```
    help_message = models.TextField(blank=True,
default="Here is how to use the bot.",
help_text="Message sent when user types /help.")

    messages = models.JSONField(default=list,
blank=True, help_text="List of received messages and
responses in JSON format.")  # JSON format

    # Timestamps
    created_at =
models.DateTimeField(auto_now_add=True)  # Bot
creation date
    updated_at = models.DateTimeField(auto_now=True)
# Last bot update date

    name = models.CharField(max_length=255,
blank=True, default="My Bot", help_text="Bot name.")
# Bot name
    description = models.TextField(blank=True,
default="This is my first bot", help_text="Bot
description.")  # Bot description

    def __str__(self):
        return f'{self.bot_type.capitalize()} Bot -
{self.user.username}'

    # Add clean method for validation
    def clean(self):
        # Validate that the bot name is not empty
        if not self.name.strip():
            raise ValidationError("Bot name cannot be
empty.")

        # Validate that bot description is not more
than 500 characters
        if len(self.description) > 500:
            raise ValidationError("Bot description
cannot exceed 500 characters.")

        # Validate Telegram token only if bot_type is
Telegram
        if self.bot_type == 'telegram':
            # Correct Telegram token pattern: digits
followed by a colon, then a random string
            if not re.match(r'^\d+:[\w-]+$',
self.token_telegram):
                raise ValidationError("Invalid
Telegram token. Ensure the token is in the correct
format like '123456:ABC-DEF1234ghIkl-
zyx57W2v1u123ew11'.")

    class Meta:
        verbose_name = "Bot"
        verbose_name_plural = "Bots"
```

Explanation of Validation Code:

1. ***Bot Name Validation***
   The bot name must be filled in and cannot be empty. By using the `strip()` function, we ensure that even if the name only contains whitespace, it will still be considered invalid. If the name is empty, an error message will appear: "Bot name cannot be empty."
2. ***Bot Description Validation***
   The bot description is limited to 500 characters. This limit is essential to ensure that the description remains concise and to the point. If the description exceeds the limit, an error message will appear: "Bot description cannot exceed 500 characters."
3. ***Telegram Token Validation***
   The Telegram token must follow a specific pattern: the token starts with several digits, followed by a colon, and then followed by a series of random characters. A correct token format example is `123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11`. If the token does not follow this pattern, an error message will

appear: "Invalid Telegram token. Ensure the token is in the correct format."

Adding Validation to Forms

Once we add validation to the model, any forms that use this model will automatically inherit the defined validations. This means that if a user tries to enter an empty bot name, an overly long description, or an invalid Telegram token, the form will provide appropriate error feedback.

Adding validation at the model level ensures that all data entering the database meets the required conditions, preventing issues that may arise later.

Implementing validation at the model level offers significant advantages in maintaining data integrity. In the Bot model, the added validation ensures that the bot name is filled, the bot description is not overly long, and the Telegram token is valid according to the correct pattern. This way, the data stored in the database is guaranteed to be valid, and users also benefit from clear and informative error messages.

In the next sub-chapter, we will discuss how to link the model, template, and form in views and how to effectively display these error messages in the user interface.

If any attribute is invalid or empty, Django will return an error, preventing the invalid data from being saved in the database.

With this model setup, we have created a strong foundation for token validation in our bot application. The next step is to implement this logic in the views and forms so that users can enter valid tokens when creating a new bot. In the next section, we will explore how to build views and forms to handle this token input effectively.

9.4 Implementing Validation In Forms

To ensure that token validation is done before form data is saved to the database, we can add validation at the form level in the `forms.py` file. This step is crucial to ensure the system receives valid data, especially regarding the format of the Telegram token and related WhatsApp information. The form will provide appropriate feedback to the user if there are errors in the data entry.

The first step is to add validation using the `clean_<field_name>()` method in the form. We will implement validation for the Telegram token using a specific pattern and also validate other fields like the bot's name and description.

9.4.1 Form-Level Validation

Django provides flexibility in data validation using `ModelForm`. In this case, we will use the `clean_token_telegram()` method to validate the Telegram token specifically. Additionally, we'll use the `widgets` attribute to enhance the form's appearance and `help_texts` to provide clear guidance to users.

Here is the detailed implementation to validate the Telegram token format before saving the data:

```
# File: apps/bots/forms.py

from django import forms
from django.core.exceptions import ValidationError
import re  # For validating token pattern
from .models import Bot  # Import the Bot model

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot  # Link the form to the Bot model
        fields = ['bot_type', 'token_telegram',
'name', 'description', 'is_active']  # Fields
displayed in the form
        widgets = {
            'name': forms.TextInput(attrs={'class':
'form-control', 'placeholder': 'Enter bot name'}),
            'description':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Bot description'}),
        }
        help_texts = {
            'token_telegram': 'Enter the unique token
received from the Telegram platform.',
            'name': 'The bot name can be changed as
needed.',
            'description': 'A brief description of
the bot and its functionality.',
        }

    # Custom validation for the Telegram token
    def clean_token_telegram(self):
        token_telegram =
self.cleaned_data.get('token_telegram')
```

```
        # Telegram token pattern: numbers followed by
a colon, then a series of random characters
        if token_telegram and not re.match(r'^\d+:[\
w-]+$', token_telegram):
            raise ValidationError("Invalid Telegram
token. Ensure the token format is like '123456:ABC-
DEF1234ghIkl-zyx57W2v1u123ew11'.")

        return token_telegram

    # Bot name validation
    def clean_name(self):
        name = self.cleaned_data.get('name')
        if not name.strip():
            raise ValidationError("The bot name
cannot be empty.")
        return name

    # Bot description validation
    def clean_description(self):
        description =
self.cleaned_data.get('description')
        if len(description) > 500:
            raise ValidationError("The bot
description cannot exceed 500 characters.")
        return description
```

Form-Level Validation Explanation:

1. ***Telegram Token Validation:*** In the `clean_token_telegram()` method, we validate the Telegram token using regex (regular expressions). The Telegram token must follow the format `bot_id:secret_key`, where `bot_id` consists of numbers, followed by a colon, and `secret_key` is a series of random characters. If the token format is incorrect, the system will display an error message: *"Invalid Telegram token. Ensure the token format is like '123456*

'."

2. ***Bot Name Validation:*** The bot's name cannot be empty or consist solely of spaces. The `clean_name()` method ensures that the bot's name contains valid characters. If not, the error message *"The bot name cannot be empty."* will appear.
3. ***Bot Description Validation:*** The bot description is limited to 500 characters. The `clean_description()` method ensures that the description does not exceed this limit. If the description is too long, the system will give an error message: *"The bot description cannot exceed 500 characters."*

By adding validation at the form level, we ensure that the data entered by users meets the expected format. Validation for the Telegram token, bot name, and bot description is effectively implemented through the `clean_<field_name>()` methods. Additionally, displaying error messages in the template helps users understand errors more easily, improving their experience.

9.5 Using Custom Validators

In addition to using the `clean_<field_name>()` method in forms, we can create custom validators in Django to check the format of tokens or other complex data. With custom validators, the validation can be applied not only in forms but also at the model level or for other fields that require the same validation. This approach makes validation more modular and reusable.

Creating Custom Validators

A custom validator is a function or class used to validate specific data before it is saved to the database. In this example, we will create two custom validators:

- ***Validator for Telegram Token***: Ensures the token follows the `bot_id:secret_key` format.
- ***Validator for Twilio Account SID***: Validates the Twilio account SID, which must consist of alphanumeric characters of a specific length.

Implementing Custom Validators

Here is how to create a custom validator for the Telegram token:

```
# File: apps/bots/validators.py

import re
from django.core.exceptions import ValidationError

# Validator for Telegram token
def validate_telegram_token(value):
    """Validate Telegram token format which should be
bot_id:secret_key."""
    if not re.match(r'^\d+:[\w-]+$', value):
        raise ValidationError(
            "Invalid Telegram token. The correct
format is '123456:ABC-DEF1234ghIkl-
zyx57W2v1u123ew11'."
        )
```

Using Custom Validators in a Model

After creating the custom validator, we can use it directly in the model. This way, every time the `Bot` model is used, the validation will automatically run when an object is created or updated.

```
# File: apps/bots/models.py

from django.db import models
from django.contrib.auth.models import User
from .validators import validate_telegram_token

class Bot(models.Model):
    BOT_CHOICES = [
        ('telegram', 'Telegram'),
        ('whatsapp', 'WhatsApp'),
    ]
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # Bot linked to a user
    bot_type = models.CharField(max_length=20,
choices=BOT_CHOICES)  # Bot type: Telegram or
WhatsApp

    token_telegram = models.CharField(
        max_length=255,
        unique=True,
        validators=[validate_telegram_token],  #
Adding validator for Telegram token
        help_text="Enter the unique bot token."
    )

    webhook_url = models.URLField(max_length=255,
blank=True, help_text="Bot's webhook URL will be set
automatically.")
    is_active = models.BooleanField(default=True,
help_text="Bot's active status.")

    start_message = models.TextField(blank=True,
default="Welcome to the bot!", help_text="Message
sent when the user types /start.")
    help_message = models.TextField(blank=True,
default="Here’s how to use the bot.",
help_text="Message sent when the user types /help.")
```

```
    messages = models.JSONField(default=list,
blank=True, help_text="List of received messages and
their responses in JSON format.")

    created_at =
models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    name = models.CharField(max_length=255,
blank=True, default="My Bot", help_text="Bot name.")
    description = models.TextField(blank=True,
default="This is my first bot", help_text="Bot
description.")

    def __str__(self):
        return f'{self.bot_type.capitalize()} Bot -
{self.user.username}'

    class Meta:
        verbose_name = "Bot"
        verbose_name_plural = "Bots"
```

Using Custom Validators in a Form

If we want to apply the validation in a form instead of a model, the custom validator can also be added to the form field. Here's an example of how to use it in a form:

```
# File: apps/bots/forms.py

from django import forms
from .models import Bot
from .validators import validate_telegram_token

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot  # Linking the form to the Bot
model
        fields = ['bot_type', 'token_telegram',
'name', 'description', 'is_active']
        widgets = {
            'name': forms.TextInput(attrs={'class':
'form-control', 'placeholder': 'Enter the bot
name'}),
            'description':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Bot description'}),
        }
        help_texts = {
            'token_telegram': 'Enter the unique token
received from the Telegram platform.',
            'name': 'The bot name can be changed as
desired.',
            'description': 'A brief description of
the bot and its function.',
        }

    # Adding form validation for Telegram token using
custom validator
    token_telegram =
forms.CharField(validators=[validate_telegram_token])
```

Validator Explanation:

1. ***Telegram Token Validator***:
    - This validator uses a regular expression (regex) to ensure the Telegram token follows the `bot_id:secret_key` pattern.
    - If the token format is incorrect, a specific error message will appear, such as: "Invalid Telegram token. The correct format is '123456
'."

By using custom validators, we can ensure that the data entered by users is valid before it is saved into the database. Custom validators offer additional flexibility because they can be reused in various places, whether in models or forms. This approach makes the validation logic more modular and helps maintain data consistency throughout the application.

9.6 Handling Token Input Errors

In systems that accept user input, such as tokens for connecting Telegram or WhatsApp bots, it's essential to handle input errors well and provide clear messages. This helps users easily correct their mistakes and avoid confusion during the bot creation process. In this section, we will discuss how to display more informative error messages and how to use Bootstrap components to make the error message presentation more user-friendly.

9.6.1 Displaying Clear Error Messages

Displaying clear error messages is the first step in handling input errors. Ambiguous or non-specific error messages can confuse users. For example, if the token entered does not match the expected format, it's crucial to display a message that specifically points out the error, such as, "Invalid token. Please ensure you're using the correct format."

Clear and informative error messages are essential during validation. When users enter an invalid or incorrect token, we must ensure they understand the reason behind the error. Using Django forms, we've already set up a mechanism to provide error messages within the `clean_token()` method. However, we also need to ensure that these messages are displayed in the user interface so they can easily see and fix the error.

To achieve this, we'll leverage Django's built-in validation mechanism, where validation errors on forms are automatically captured and stored in the `form.errors` attribute. These errors are then displayed in the template using a loop for each field that encountered an error.

When the form is submitted and validation fails, Django automatically adds error messages to the form object. In the template, we can display these messages using a loop to iterate through each field.

In Django forms, error messages are generated by the existing validation mechanism. When a user fills out the form and validation fails, the errors are stored in the `form.errors` attribute. To display these errors in the user interface, we can use Bootstrap elements such as `invalid-feedback`.

Here's an example of displaying token validation errors in the template using Bootstrap elements:

```
<!-- File: apps/templates/bots/create_bot.html -->
<div class="form-group" id="token-field">
    <label for="{{ form.token.id_for_label
}}">Token</label>
    <input type="text" name="token"
id="{{ form.token.id_for_label }}"
value="{{ form.token.value }}" class="form-control {%
if form.token.errors %}is-invalid{% endif %}">
    {% for error in form.token.errors %}
        <div class="invalid-feedback">{{ error
}}</div>
    {% endfor %}
</div>
```

In the code above, when the token is invalid, the field receives the `is-invalid` class, which automatically highlights the input field in red according to Bootstrap styles. Error messages appear below the input field with the `invalid-feedback` class, ensuring a professional and clean look.

9.6.2 Using Bootstrap for Error Notifications

To make the error messages more user-friendly and professional, we can leverage Bootstrap components such as `invalid-feedback` for field-specific errors and Bootstrap modals to display more comprehensive error notifications for larger issues.

Bootstrap offers various components that can enhance the presentation of error messages. By using Bootstrap classes, we can display error messages in the form of clear and attractive notifications. For instance, we can wrap error messages inside a `div` element with the appropriate `alert` class.

Here's an example of using a Bootstrap modal to display error messages when the form is invalid:

```
{% if form.errors %}
<div id="errorModal" class="modal fade" tabindex="-1"
aria-labelledby="errorModalLabel" aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title"
id="errorModalLabel">An Error Occurred</h5>
        <button type="button" class="btn-close" data-
bs-dismiss="modal" aria-label="Close"></button>
      </div>
      <div class="modal-body">
        {% for field in form %}
```

```
            {% for error in field.errors %}
                <p>{{ error }}</p>
            {% endfor %}
        {% endfor %}
        {% for error in form.non_field_errors %}
            <p>{{ error }}</p>
        {% endfor %}
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-
secondary" data-bs-dismiss="modal">Close</button>
      </div>
    </div>
  </div>
</div>
{% endif %}
```

This modal will appear if the form is invalid, showing all error messages in a single view. To automatically trigger the modal when form validation fails, we can add a small JavaScript snippet:

```
document.addEventListener('DOMContentLoaded',
function () {
    if (document.querySelector('#errorModal')) {
        var errorModal = new
bootstrap.Modal(document.getElementById('errorModal')
);
        errorModal.show();
    }
});
```

With this approach, any error not specific to a single field, such as system errors or more general issues, can be clearly presented to the user. The modal also provides a more professional and user-friendly display compared to simply showing the error on the page.

By displaying clear error messages and leveraging Bootstrap components like `invalid-feedback` and modals, we can significantly improve the user experience. Effective error handling helps reduce confusion and ensures the bot creation process runs smoothly.

Using Bootstrap components like these also helps maintain a consistent and user-friendly application interface. Well-handled input errors with an appealing design can reduce user frustration, especially when dealing with sensitive inputs like tokens.

Thus, this section has highlighted the importance of providing clear error messages and how we can use Bootstrap to make error notifications more user-friendly.

9.7 Complete Code After Validation

We can choose to perform validation in one place depending on the application's needs and workflow. However, there are several reasons why we might want to use either or both methods:

Validation in Models:

- ***Advantages:*** Validation in the model ensures that every time an object is saved to the database, validation is executed. This is useful if the model is used in places other than forms, such as APIs or custom scripts.

- ***Disadvantages:*** Validation in the model does not provide clean error messages on the HTML form, as this validation is more internal.

Validation in Forms:

- ***Advantages:*** Validation in forms allows us to give direct feedback to users via the form with user-friendly and easy-to-understand error messages, which are displayed directly on the page.
- ***Disadvantages:*** Validation in forms only applies when data is submitted through a form. If data is entered into the model outside of the form (e.g., via an API), this validation will not run.

So, What’s the Best Approach?

If we only want to ensure validation when users enter data through the form, validation in forms is sufficient. However, if we want validation to run every time the model object is saved, regardless of how the data is entered, it is best to place validation in models.

Option: Choose One

If we decide to perform validation only in forms, the validation in the models can be removed. Conversely, if validation only exists in models, the form will simply call `full_clean()`, which will execute the validation in the model.

Example of validation if only done in forms:

1. ***Remove validation in `models.py`:***

```
def clean(self):
    # Validation to ensure token uniqueness
    if
Bot.objects.exclude(id=self.id).filter(token=s
elf.token).exists():
        raise ValidationError('A bot with this
token already exists.')
```

2. ***Keep validation in `forms.py`:***

```
def clean_token(self):
    token = self.cleaned_data.get('token')
    bot_type =
self.cleaned_data.get('bot_type')

    # Validate Telegram and WhatsApp tokens in
the form
    if bot_type == 'telegram':
        validate_telegram_token(token)
    elif bot_type == 'whatsapp':
        validate_whatsapp_token(token)

    # Validate unique token
    if
Bot.objects.exclude(id=self.instance.id).filte
r(token=token).exists():
        raise forms.ValidationError('A bot
with this token already exists.')

    return token
```

With validation in the form, error messages will be displayed directly to the user on the submitted form page.

- ***Validation in models:*** Safer for all data storage workflows, including APIs and scripts.
- ***Validation in forms:*** More user-friendly for validation shown on the web page.

If validation is only required when users fill out the form on the web page, place validation in forms. However, if validation should apply across the entire application, it's better to place it in models.

9.7.1 Models.py

Below is the updated `Bot` model code with added validation for token format based on the bot type, if we choose to use validation in the model:

```
# File: apps/bots/models.py

from django.db import models
from django.contrib.auth.models import User
from django.core.exceptions import ValidationError  #
Import for validation
import re  # Import for validating token patterns

class Bot(models.Model):
    BOT_CHOICES = [
        ('telegram', 'Telegram'),
        ('whatsapp', 'WhatsApp'),
    ]
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # Bot will be associated
with a user
    bot_type = models.CharField(max_length=20,
choices=BOT_CHOICES)  # Bot type: Telegram or
WhatsApp

    token_telegram = models.CharField(max_length=255,
unique=True, help_text="Enter a unique bot token.")
# Telegram token
    webhook_url = models.URLField(max_length=255,
blank=True, help_text="Bot webhook URL will be set
automatically.")  # Webhook URL
```

```
    is_active = models.BooleanField(default=True,
help_text="Bot's active status.")  # Active status of
the bot

    # Auto-messages
    start_message = models.TextField(blank=True,
default="Welcome to the bot!", help_text="Message
sent when user types /start.")
    help_message = models.TextField(blank=True,
default="Here's how to use the bot.",
help_text="Message sent when user types /help.")

    messages = models.JSONField(default=list,
blank=True, help_text="List of received messages and
their responses in JSON format.")  # JSON format

    # Timestamps
    created_at =
models.DateTimeField(auto_now_add=True)  # Bot
creation date
    updated_at = models.DateTimeField(auto_now=True)
# Last updated date of the bot

    name = models.CharField(max_length=255,
blank=True, default="My Bot", help_text="Bot name.")
# Bot name
    description = models.TextField(blank=True,
default="This is my first bot", help_text="Bot
description.")  # Bot description

    def __str__(self):
        return f'{self.bot_type.capitalize()} Bot -
{self.user.username}'

    # Add a clean method for validation
    def clean(self):
        # Validate that bot name is not empty
        if not self.name.strip():
            raise ValidationError("Bot name cannot be
empty.")

        # Validate bot description is no longer than
500 characters
```

```
        if len(self.description) > 500:
            raise ValidationError("Bot description
cannot exceed 500 characters.")

        # Validate Telegram token only if bot_type is
Telegram
        if self.bot_type == 'telegram':
            # Correct pattern for Telegram token:
digits followed by a colon, then a string of random
characters
            if not re.match(r'^\d+:[\w-]+$',
self.token_telegram):
                raise ValidationError("Invalid
Telegram token. Ensure the token is in the correct
format like '123456:ABC-DEF1234ghIkl-
zyx57W2v1u123ew11'.")

    class Meta:
        verbose_name = "Bot"
        verbose_name_plural = "Bots"
```

With this implementation, the `Bot` model now has more robust logic to handle token validation based on the bot type selected by the user.

9.7.2 Forms.py

Below is the updated code for the ***BotForm*** form with token validation implementation based on the bot type. If you want to add validation in forms:

```
# File: apps/bots/forms.py

from django import forms
from django.core.exceptions import ValidationError
import re  # For validating token pattern
from .models import Bot  # Importing the Bot model

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot  # Linking the form to the Bot
model
        fields = ['bot_type', 'token_telegram',
'name', 'description', 'is_active', 'start_message',
'help_message']  # Fields displayed in the form
        widgets = {
            'name': forms.TextInput(attrs={'class':
'form-control', 'placeholder': 'Enter bot name'}),
            'description':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Bot description'}),
        }
        help_texts = {
            'token_telegram': 'Enter the unique token
received from the Telegram platform.',
            'name': 'The bot name can be changed as
needed.',
            'description': 'A brief description of
the bot and its functions.',
        }

    # Custom validation for Telegram token
    def clean_token_telegram(self):
        token_telegram =
self.cleaned_data.get('token_telegram')
        bot_type = self.cleaned_data.get('bot_type')

        # Validation only if bot_type is Telegram
        if bot_type == 'telegram':
            # Telegram token pattern: digits followed
by a colon, then a series of random characters
            if token_telegram and not re.match(r'^\
d+:[\w-]+$', token_telegram):
                raise ValidationError("Invalid
Telegram token. Ensure the token format is like
'123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11'.")

        return token_telegram

    # Bot name validation
    def clean_name(self):
        name = self.cleaned_data.get('name')
        if not name.strip():
            raise ValidationError("Bot name cannot be
empty.")
        return name

    # Bot description validation
    def clean_description(self):
        description =
self.cleaned_data.get('description')
        if len(description) > 500:
            raise ValidationError("Bot description
cannot exceed 500 characters.")
        return description
```

With this implementation, the ***BotForm*** can now properly validate tokens based on the bot type selected by the user.

9.7.3 Create_bot.html

Below is the updated ***create_bot.html*** code with error handling and a cleaner form layout using Bootstrap:

```
<!-- File: apps/templates/bots/create_bot.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <!-- Card for Form -->
    <div class="card">
        <div class="card-header">
            <h5 class="card-title">Create New
Bot</h5>
        </div>
        <div class="card-body">
            <!-- Bot Form -->
            <form method="post">
                {% csrf_token %}

                <!-- Form Group for Platform (appears
first) -->
                <div class="form-group">
                <label class="form-label">
                    Platform
                </label>
                <div class="btn-group"
role="group" aria-label="Platform">
                    <input type="radio"
name="bot_type" id="telegram" value="telegram" {% if
form.bot_type.value == 'telegram' %}checked{% endif
%} onclick="toggleFields()">
                    <label
for="telegram">Telegram</label>

                    <input type="radio"
name="bot_type" id="whatsapp" value="whatsapp" {% if
form.bot_type.value == 'whatsapp' %}checked{% endif
%} onclick="toggleFields()">
                    <label
for="whatsapp">WhatsApp</label>
                </div>
            </div>

            <!-- Form Group for Name -->
            <div class="form-group">
                <label
for="{{ form.name.id_for_label }}">
                    Name
                </label>
                <input type="text" name="name"
id="{{ form.name.id_for_label }}"
value="{{ form.name.value }}" class="form-control {%
if form.name.errors %}is-invalid{% endif %}">
                {% for error in form.name.errors
%}
                    <div class="invalid-
feedback">{{ error }}</div>
                {% endfor %}
            </div>

            <!-- Form Group for Token (only for
Telegram) -->
            <div class="form-group" id="token-
field">
```

```
                    <label
for="{{ form.token_telegram.id_for_label }}">
                        Token
                    </label>
                    <input type="text"
name="token_telegram"
id="{{ form.token_telegram.id_for_label }}" value="{{
form.token_telegram.value }}" class="form-control {%
if form.token_telegram.errors %}is-invalid{% endif
%}">
                    {% for error in
form.token_telegram.errors %}
                        <div class="invalid-
feedback">{{ error }}</div>
                    {% endfor %}
                </div>

                <!-- Submit Button -->
                <button type="submit" class="btn btn-
primary">
                    Create Bot
                </button>
            </form>
        </div>
    </div>
    <!-- Error Message Modal -->
    {% if form.errors %}
    <div class="modal fade" id="errorModal"
tabindex="-1" aria-labelledby="errorModalLabel" aria-
hidden="true">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header bg-danger
text-white">
                    <h1 class="modal-title fs-5"
id="errorModalLabel">An Error Occurred</h1>
                    <button type="button" class="btn-
close" data-bs-dismiss="modal" aria-
label="Close"></button>
                </div>
                <div class="modal-body">
                    <div class="alert alert-danger">
                        {% for field in form %}
```

```
                            {% for error in
field.errors %}
                                <p>{{ error }}</p>
                            {% endfor %}
                        {% endfor %}
                        {% for error in
form.non_field_errors %}
                            <p>{{ error }}</p>
                        {% endfor %}
                    </div>
                </div>
                <div class="modal-footer">
                    <button type="button" class="btn
btn-secondary" data-bs-dismiss="modal">Close</button>
                </div>
            </div>
        </div>
    </div>

    <script>
        // Ensure Bootstrap JavaScript is loaded and
jQuery if needed
        document.addEventListener('DOMContentLoaded',
function () {
            if
(document.querySelector('#errorModal')) {
                var errorModal = new
bootstrap.Modal(document.getElementById('errorModal')
);
                errorModal.show();
            }
        });
    </script>
    {% endif %}
</div>

<script>
$(document).ready(function() {
    // Show or hide the token field based on the
selected bot type
    $('#id_bot_type').change(function() {
        var botType = $(this).val();
        if (botType === 'whatsapp') {
            $('#id_token_telegram').closest('.form-
group').hide();  // Hide Telegram token field
        } else {
            $('#id_token_telegram').closest('.form-
group').show();  // Show Telegram token field
        }
    }).trigger('change');  // Trigger event on page
load
});
</script>
<script>
    // Function to hide/show elements based on the
platform
    function toggleFields() {
        var tokenField =
document.getElementById('token-field');
        var whatsappFields =
document.getElementById('whatsapp-fields');

        if
(document.getElementById('telegram').checked) {
            tokenField.classList.remove('d-none');
            whatsappFields?.classList.add('d-none');
        } else if
(document.getElementById('whatsapp').checked) {
            tokenField.classList.add('d-none');
            whatsappFields?.classList.remove('d-
none');
        }
    }

    // Call toggleFields() when the page loads
    document.addEventListener('DOMContentLoaded',
function () {
        toggleFields();
    });
</script>
{% endblock %}
```

With this implementation, the form layout is cleaner, and users can clearly see input errors. Ensure the `add_class` function is loaded in the template filter to dynamically add Bootstrap classes.

Chapter Conclusion

This chapter has discussed the importance of token validation in bot creation, focusing on how to ensure that the tokens entered by users are valid and adhere to the expected format. Token validation not only enhances application security but also improves user experience by preventing errors that may occur due to incorrect tokens.

We have explored various types of tokens and their usage, as well as why the correct token format is crucial. Additionally, we have implemented token validation in the context of a Django application, from creating the appropriate model to handling input through forms. By using validation methods at the form level, we can ensure that only valid tokens are stored in the database.

In handling input errors, we emphasized the importance of providing clear feedback to users. By displaying informative error messages and using Bootstrap components for error notifications, we improve both readability and the aesthetics of the user interface.

With the steps discussed in this chapter, our application is now more robust and ready to handle various user inputs securely and efficiently. Next, we will move on to other topics related to bot management and additional functionalities that can be incorporated into our application.

Chapter 10 - Setting Up Webhooks for Bots

In the world of bot development, webhooks play a crucial role as a bridge between bot platforms and our application server. Webhooks enable our application to receive real-time notifications from bot platforms when various events occur, such as new messages or status updates.

By using webhooks, our application can automatically process data and respond to user interactions without needing to continuously poll the server. In this chapter, we will discuss the steps to set up a webhook in a Django project named `platform_bot`, allowing us to handle and respond to data sent by the bot platform.

10.1 Introduction To Webhooks

Webhooks are a fundamental concept in modern application development, especially when dealing with real-time interactions between different systems. A webhook is a mechanism that allows one application to automatically send data to another application when certain events occur. In the context of bots, webhooks serve as a communication bridge between bot platforms (like Telegram or WhatsApp) and your server.

10.1.1 What Is a Webhook?

A webhook is a URL endpoint set up to receive real-time data from an external platform. When a relevant event occurs on that platform—such as receiving a new message in a bot—the platform will send the event data to the webhook URL you've specified. This allows your server to directly respond to the event without needing to actively request data periodically.

For example, if you have a Telegram bot and want it to respond to messages sent by users, you will configure a webhook for the bot. When a user sends a message to the bot, Telegram will send the message data to your webhook endpoint. Your server will then process the data and execute the necessary logic, such as sending a reply or storing the information in a database.

A webhook is a way to provide web applications with information in real-time. Unlike the polling method, where the application periodically checks the server for updates, webhooks allow the server to notify the application automatically when new data is available. This not only reduces server load but also makes our application more responsive to user interactions.

By using webhooks, we can simplify integration with bot platforms. For instance, when a user sends a message to the bot, the webhook will send the message data to our server without waiting for a request from the server. This enables our bot to respond to messages quickly and efficiently.

10.1.2 Benefits of Using Webhooks

Using webhooks offers several significant benefits, particularly in terms of efficiency and responsiveness. One of the main advantages is resource savings. Without webhooks, your application would need to continuously check (poll) the external platform to see if there are any updates or new events. This can put a strain on your server and inefficiently consume bandwidth.

With webhooks, your server only receives data when there is a relevant change or event, thus reducing server load and improving communication efficiency. Moreover, webhooks enable your application to respond to events in real-time, which is crucial for applications like bots that require fast interactions with users.

10.1.3 How Webhooks Work

The way webhooks work can be explained through a few key steps. First, you need to set up a webhook URL on the external platform (e.g., Telegram). This URL will serve as the endpoint on your server that will receive data. When a certain event occurs on the external platform, the data will be automatically sent to the webhook URL you've registered.

On your server, the webhook endpoint will receive an HTTP POST request containing the event data. For instance, in the case of a Telegram bot, this data might include the message sent by a user. Once the data is received, your server will process it according to your application's logic. This could involve saving the data to a database, processing commands, or sending a reply to the user.

10.1.4 Setting Up Webhooks in Django

To integrate a webhook into your Django project, you will need to set up a few key components. First, you must create a Django view that can handle the requests sent to the webhook URL.

Next, you will add URL routing that directs the requests to the appropriate view. Finally, you will need to test the integration to ensure that the webhook is working as expected.

This process will enable your server to interact with various bot platforms through a single webhook, making bot management simpler and more efficient.

With this basic understanding of webhooks, you are ready to proceed to the next step of implementing a webhook in your Django project. Next, we will discuss how to create a view for the webhook, set up URL routing, and test the webhook integration to ensure everything is functioning correctly.

10.2 Structuring And Managing Webhooks

In implementing a webhook that supports multiple platforms like Telegram and WhatsApp, it’s essential to design a flexible system that can handle data from various sources. A single universal webhook can be used for multiple platforms if properly designed, so you don’t have to create separate webhooks for each platform.

To achieve this, your webhook must be capable of processing data from various platforms with different formats. This involves recognizing and processing the different data structures from each platform at a single endpoint. For example, the data sent by Telegram differs from the data sent by WhatsApp, but you can set up a system to process both.

As an example, you can use a Django view that identifies the data source based on the information received, such as headers or parameters in the request. With this approach, a single webhook endpoint can be used to receive and process data from different platforms.

10.2.1 Handling Data from Webhooks

Once the webhook is set up, you need to handle the data received from different platforms. The data received from a webhook is usually in JSON format, and each platform may have a different data structure.

For example, Telegram data typically contains a JSON object with information about the message, the sender, and more. WhatsApp data might have a different structure, such as information about text or media messages. To process this data, you need to write code that can identify and handle different data formats.

By doing this, you can manage a universal webhook that handles data from multiple platforms within a single Django application. Next, we will discuss how to configure URL routing for the webhook and test the integration to ensure everything functions as expected.

10.3 Setting Up A Telegram Webhook

Now that we've understood the basics of webhooks in the previous section, we will proceed with preparing the webhook for the Telegram platform. This webhook will enable the Telegram bot to automatically receive and process incoming messages. To achieve this, we need to ensure that our application can handle updates sent by Telegram correctly.

10.3.1 Configuring the Telegram Webhook

The first step in setting up the Telegram webhook is creating an endpoint that will receive and process updates sent by Telegram's server. Telegram will send JSON data to the URL we set as the webhook, and our task is to process and respond to this data appropriately. To handle this webhook, we will create a specific Django view that will handle requests from Telegram.

Here's the code:

```
# File: apps/webhook/views.py

import json
import requests
```

```
from django.http import HttpResponse,
HttpResponseBadRequest
from django.views.decorators.csrf import csrf_exempt
import logging
from apps.bots.services.telegram import handle_update
as handle_telegram_update

# Logger to record information and errors
logger = logging.getLogger(__name__)

@csrf_exempt  # Disable CSRF protection for this
endpoint
def telegram_webhook(request):
    if request.method == 'POST':  # Ensure the
request is a POST
        try:
            # Load JSON data from the request body
            update =
json.loads(request.body.decode('utf-8'))
            logger.info(f"Telegram update received:
{update}")  # Log the received update

            # Ensure there's a 'message' in the
received update
            if 'message' in update:
                handle_telegram_update(update)  #
Process the update through the service created
                return HttpResponse('ok')  # Return
HTTP 200 OK

            # If there's no 'message' in the update,
return HTTP 400 Bad Request
            return HttpResponseBadRequest('No message
found in update')

        except json.JSONDecodeError:  # Handle errors
if the JSON is invalid
            logger.error("Invalid JSON received.")
            return HttpResponseBadRequest('Invalid
JSON format')

        except Exception as e:  # Handle any
unexpected errors
```

```
            logger.error(f"Error processing Telegram
webhook: {e}")
            return HttpResponse('Error', status=500)

    # If the request is not a POST, return HTTP 400
Bad Request
    return HttpResponseBadRequest('Bad Request',
status=400)
```

In the code above, we set up a view that will handle POST requests from Telegram. When Telegram sends an update to our webhook, our server will receive the data in JSON format. In this JSON, we need to ensure that the data is valid, particularly that there is a message sent (a 'message'). If a message is found, the update will be processed by the `handle_telegram_update` function, which is part of our Telegram bot service. If no message is found, we return an HTTP 400 response, indicating that there was an issue with the received request. We also ensure that any errors that occur are logged using Python's logging module.

10.3.2 Configuring the Telegram Webhook

After creating the endpoint that handles updates, the next step is to configure the webhook on Telegram’s server. This involves calling the Telegram API to link our application’s URL with the bot that has been created. To do this, we need to use the bot token registered on Telegram's platform and the URL of our application, which will be used as the webhook.

Here is the code to set up the Telegram webhook:

```
# File: apps/webhook/views.py
```

```
def set_telegram_webhook(token_telegram, url):
    # Configure the Telegram API URL with the bot
token
    TELEGRAM_API_URL =
f'https://api.telegram.org/bot{token_telegram}/'

    # Send a request to the Telegram API to set the
webhook
    response = requests.post(TELEGRAM_API_URL +
"setWebhook?url=" + url).json()

    # Check if the response from Telegram indicates
success
    if not response.get('ok'):
        logger.error(f"Failed to set Telegram
webhook: {response}")  # Log if it fails
    else:
        logger.info(f"Telegram webhook set
successfully: {response}")  # Log if it succeeds

    return response
```

In the function above, we call the Telegram API to set the webhook. We provide the registered bot token and our application's URL, which serves as the webhook endpoint. If the webhook is successfully set, Telegram returns a response indicating the operation was successful, and we log that information. If there's an error setting the webhook, we also log the error.

10.4 Setting Up The WhatsApp Webhook

After setting up the Telegram webhook, the next step is to configure the webhook for the WhatsApp platform. Unlike Telegram, WhatsApp webhooks use third-party services like

Twilio to send and receive messages. Therefore, we need to ensure a smooth integration between Twilio and our application.

10.4.1 Configuring the Webhook for WhatsApp

Just like with Telegram, we need to create an endpoint in Django that will handle updates from WhatsApp. Twilio sends data in the form of messages and related information about the sender, usually as form data rather than JSON.

Here’s the code to configure the WhatsApp webhook:

```
# File: apps/webhook/views.py

import json
import requests
from django.http import HttpResponse,
HttpResponseBadRequest
from django.views.decorators.csrf import csrf_exempt
import logging
from apps.bots.services.whatsapp import handle_update
as handle_whatsapp_update

# Logger to record information and errors
logger = logging.getLogger(__name__)

@csrf_exempt  # Disabling CSRF protection for this
endpoint
def whatsapp_webhook(request):
    if request.method == 'POST':  # Ensure the
request is POST
        try:
            # Twilio sends data as form-urlencoded,
we extract it from request.POST
            if 'From' in request.POST and 'Body' in
request.POST:
                # Create an update structure based on
the required data
                update = {
                    'From': request.POST['From'],  # 
Sender's number
                    'Body': request.POST['Body']   # 
Message content
                }

                # Process the update using the 
WhatsApp service
                return handle_whatsapp_update(update)

            # If no 'From' and 'Body' data, return 
HTTP 200 OK
            return HttpResponse('ok')

        except Exception as e:  # Handle any errors 
that might occur
            logger.error(f"Error processing WhatsApp 
webhook: {e}")
            return HttpResponseBadRequest('Failed to 
process update')  # Return HTTP 400 if there's an 
error

    # If the request is not POST, return HTTP 400 Bad 
Request
    return HttpResponseBadRequest('Bad Request')
```

In the code above, we create a view that receives POST requests from Twilio, where WhatsApp messages are sent as form-urlencoded data. The main data we need is the `From` (sender's number) and the `Body` (message content). If both pieces of data are present, we form an update that will be processed by the `handle_whatsapp_update` function, which is set up in the WhatsApp bot service. If the data is not appropriate, we still return a 200 OK response to avoid further errors from Twilio’s side.

This function also logs any errors that might occur during the webhook process, making it easier to track issues if they arise.

10.4.2 Configuring WhatsApp Webhook through Twilio

Unlike Telegram, for WhatsApp, we cannot directly set up the webhook via API. Instead, we need to configure this through the Twilio dashboard. Twilio provides an interface to link a registered WhatsApp number with our application’s URL, which will handle the webhook. The following code serves only as a reminder that the webhook configuration must be done via Twilio:

```
# File: apps/webhook/views.py

def set_whatsapp_webhook(url):
    # Informing that webhook must be set through
Twilio dashboard
    logger.info(f"WhatsApp webhook must be configured
through the Twilio dashboard at URL: {url}")
    return {"status": "Webhook configuration needed
via Twilio dashboard."}
```

This function logs that the WhatsApp webhook cannot be configured directly via API as in Telegram; instead, it must be manually set through the Twilio dashboard. Twilio requires the webhook URL that will be used to send incoming messages to our application.

10.4.3 Linking Webhook URL to Twilio

Once we have the URL that will handle WhatsApp webhooks, we need to link it to the registered WhatsApp number in Twilio.

Follow these steps through the Twilio dashboard:

1. Log into your Twilio account on their dashboard.
2. Select the registered WhatsApp number.
3. In the "Messaging" section, add our application's URL in the "Webhook" field.
4. Ensure the request method selected is POST.
5. Save the changes.

Once this is set, any messages sent to the registered WhatsApp number in Twilio will be forwarded to the webhook URL we've specified in our Django application.

10.4.4 Running and Testing the Webhook

After the webhook is configured via the Twilio dashboard, we can test if the webhook is working correctly. Twilio will send incoming messages to the webhook URL, and our application will process the messages using the `whatsapp_webhook` function we previously created.

To check whether the webhook is functioning correctly, we can view the application logs to ensure that the messages from Twilio are being processed successfully. If any errors occur, Twilio will also provide detailed error information through their dashboard, making troubleshooting easy.

At this point, the WhatsApp webhook is ready to use, and our application can now receive and process incoming WhatsApp messages. This allows our bot to automatically respond to users in real-time, just like on the Telegram platform.

10.5 Complete Code For Views.py

Here is the complete code with some minor improvements:

```
# File: apps/webhook/views.py

import json
import requests
from django.http import HttpResponse,
HttpResponseBadRequest
from django.views.decorators.csrf import csrf_exempt
import logging
from apps.bots.services.telegram import handle_update
as handle_telegram_update
from apps.bots.services.whatsapp import handle_update
as handle_whatsapp_update

logger = logging.getLogger(__name__)

@csrf_exempt
def telegram_webhook(request):
    if request.method == 'POST':
        try:
            update =
json.loads(request.body.decode('utf-8'))
            logger.info(f"Telegram update received:
{update}")

            if 'message' in update:
                handle_telegram_update(update)
                return HttpResponse('ok')  # Ensure a
valid response is returned

            return HttpResponseBadRequest('No message
found in update')
        except json.JSONDecodeError:
            logger.error("Invalid JSON received.")
            return HttpResponseBadRequest('Invalid
JSON format')
```

```
        except Exception as e:
            logger.error(f"Error processing Telegram
webhook: {e}")
            return HttpResponse('Error', status=500)
    return HttpResponseBadRequest('Bad Request',
status=400)


@csrf_exempt
def whatsapp_webhook(request):
    if request.method == 'POST':
        try:
            # If the payload is form-urlencoded,
retrieve it from request.POST
            if 'From' in request.POST and 'Body' in
request.POST:
                update = {
                    'From': request.POST['From'],
                    'Body': request.POST['Body']
                }
                return handle_whatsapp_update(update)

            return HttpResponse('ok')

        except Exception as e:
            logger.error(f"Error processing webhook:
{e}")
            return HttpResponseBadRequest('Failed to
process update')
    else:
        return HttpResponseBadRequest('Bad Request')


def set_telegram_webhook(token_telegram, url):
    TELEGRAM_API_URL =
f'https://api.telegram.org/bot{token_telegram}/'
    response = requests.post(TELEGRAM_API_URL +
"setWebhook?url=" + url).json()

    if not response.get('ok'):
        logger.error(f"Failed to set Telegram
webhook: {response}")
    else:
        logger.info(f"Telegram webhook set
successfully: {response}")

    return response

def set_whatsapp_webhook(url):
    logger.info(f"WhatsApp webhook must be configured
through the Twilio dashboard at URL: {url}")
    return {"status": "Webhook configuration needed
via Twilio dashboard."}
```

With this structure, you can easily add support for other platforms like Facebook Messenger, Slack, etc.

10.6 Configuring URL Routing

To make the view accessible via a URL, we need to configure URL routing in Django. Add the endpoint URLs that will be used by the webhook to send data.

Next, we need to add URL routing for the webhook so that Django knows how to handle incoming requests. Add the following URL configuration to the `urls.py` file in the webhook application:

```
# File: apps/webhook/urls.py

from django.urls import path
from .views import telegram_webhook, whatsapp_webhook

urlpatterns = [
    # Webhook endpoint for Telegram
    path('telegram/', telegram_webhook,
name='telegram_webhook'),
    # Webhook endpoint for WhatsApp
    path('whatsapp/', whatsapp_webhook,
name='whatsapp_webhook'),
]
```

Next, we need to ensure that the webhook application's URLs are included in the main project routing. Open the `platform_bot/urls.py` file and add the following configuration:

```
# File: platform_bot/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('apps.webhook.urls')),  #
Include webhook application URLs
]
```

With this configuration, the webhook page will be displayed when users visit the root URL of the application.

10.7 Configuration Of Settings.py

To test the webhook integration in a Django project, we need a public URL accessible by bot platforms such as Telegram or WhatsApp. One way to get a temporary public URL is by using a tool like ngrok. Ngrok allows us to create a tunnel to our local server and obtain a URL that can be accessed from the internet.

Running Ngrok

First, run the following command in the terminal to start ngrok and create a tunnel to the local server running on port 8000:

```
$ ngrok http 8000
```

This command will provide a temporary URL that can be accessed from the internet. Ngrok will display this URL in the terminal, such as `https://ngrok-random_id.app`. Use this URL for webhook configuration.

Configuring settings.py

Next, we need to update the `settings.py` file to include the webhook URL and set the required `ALLOWED_HOSTS`. Here are the steps to follow:

Setting ALLOWED_HOSTS

The `ALLOWED_HOSTS` parameter in Django is used to define a list of host/domain names that are allowed to access the Django application. When `ALLOWED_HOSTS` is set to `['*']`, it means the Django application will accept requests from all hosts.

This setting is typically used during development to avoid CORS (Cross-Origin Resource Sharing) issues or other problems that may arise when only using a local domain. However, in a production environment, `ALLOWED_HOSTS` should be set to specific domains or IPs to prevent unwanted access.

Update `ALLOWED_HOSTS` as follows:

```
# File: platform_bot/settings.py

ALLOWED_HOSTS = ['*']  # Allow all hosts, used during
development
```

Sending POST Request and Testing the Webhook

After configuring `ALLOWED_HOSTS`, we can test the webhook integration. Send a POST request to the `/getpost/` URL using platforms like Telegram or WhatsApp to ensure that the webhook is receiving and processing updates correctly.

With these steps, we have successfully set up a webhook in our Django project. Ngrok provides a temporary URL that allows us to test our application from outside, while the `ALLOWED_HOSTS` setting ensures the application can receive requests from all sources during development.

In the next section, we will discuss how to integrate the bot with Django and manage the data received from the webhook, continuing the process to ensure the application functions well in real-world scenarios.

Chapter Conclusion

In this section, we have thoroughly discussed webhooks, which are a crucial step in ensuring that bots can respond to updates or messages from various platforms. Starting with the ***Introduction to Webhooks***, we understood the importance of webhooks in connecting external applications with bots, enabling real-time and automated communication.

Next, in the ***Setting Up Webhook Views*** section, we learned how to structure and manage webhooks effectively to handle various types of updates from platforms like Telegram and WhatsApp. This process includes setting up webhooks for each platform, managing different payloads.

In the subsection ***Handling Updates from Various Platforms***, we created handlers for the bot to handle different types of incoming data, such as messages from Telegram and notifications from WhatsApp. This integration is managed through webhook functions organized in the `views.py` file.

Configuring Webhooks for Different Platforms discussed how to connect bots with platforms through their respective APIs, where we learned how to set up webhooks for Telegram and WhatsApp through the appropriate endpoints. This section also involved configuring URL routing in Django via `urls.py`, ensuring that requests from platforms are correctly handled by the server.

In the subsection ***Configuring settings.py***, we covered the necessary Django server settings to support webhooks, including the URL configuration required to ensure that the bot functions properly.

By completing this section, you now understand the principles and best practices for managing webhooks for bots. This chapter also provides a solid technical foundation to continue integrating bots on various platforms and enhance their capabilities in handling real-time communication.

Chapter 11 - Bot Integration with Webhook

In this chapter, we will discuss the integration of webhooks with bots, starting from the definition and implementation of webhook integration with bots so that we can create bots and have them operate using the webhooks we have set up.

The integration between bots and webhooks is a crucial step in bot-based application development. This process allows bots to receive and send data in real time, interact with users, and manage commands more efficiently. By understanding how to integrate bots with webhooks, we can fully leverage the potential of the platforms we are using, such as Telegram and WhatsApp.

11.1 Introduction To Webhook Integration With Bots

When building bots for various platforms like Telegram or WhatsApp, one crucial component to consider is webhook integration. Webhooks serve as a communication mechanism between your server and the platform where your bot operates. Integrating webhooks allows your bot to receive real-time updates or notifications from these platforms, such as new messages or user commands.

11.1.1 What is Webhook Integration?

Webhook integration is the process of connecting your application or system with external services through a specified URL. In the context of bots, webhooks are used to send information from the platform to your server whenever a relevant event occurs. For example, when a user sends a message to the bot on Telegram, the webhook sends that message data to the webhook URL you configured on your server.

Webhook integration refers to setting up communication between the bot and the platform via a specific URL that acts as an endpoint. When an event occurs on the platform, such as receiving a message or issuing a command, the platform sends data to the specified webhook URL. In the context of bots, webhooks act as a bridge that connects the bot with the platform, allowing the bot to receive updates and respond to user actions quickly.

Webhooks work on a push rather than pull principle, where the platform automatically sends data to the bot without requiring a request from the bot side. This is highly advantageous, as it reduces latency and ensures that the bot always receives the latest information. For example, when a user sends a message to the bot on Telegram, the platform sends that update to the specified webhook URL. The bot can then process this information and provide an appropriate response.

Essentially, webhooks are HTTP callbacks that allow your server to automatically receive data from other platforms without needing to poll continuously. When a platform sends updates to your webhook URL, your server can process the information and respond as needed. This is especially useful for bots that require real-time interaction with users.

To set up a webhook, we need to specify the URL that the platform will use to send data. In our Django project, this configuration is done in the `settings.py` file. After the webhook URL is set, we need to implement logic in the views to handle updates received from various platforms. Below is an example code for setting up a Telegram webhook in `views.py`:

```
# File: apps/webhook/views.py

import requests

def set_telegram_webhook(token_telegram, url):
    TELEGRAM_API_URL =
f'https://api.telegram.org/bot{token_telegram}/'
    response = requests.post(TELEGRAM_API_URL +
"setWebhook?url=" + url).json()

    if not response.get('ok'):
        logger.error(f"Failed to set Telegram
webhook: {response}")
    else:
        logger.info(f"Telegram webhook set
successfully: {response}")

    return response
```

In the above example, we set the webhook by sending a request to the Telegram API, providing the URL that will receive updates. With these steps, we ensure that our bot is properly connected and can function optimally.

With an understanding of the basic concept of webhook integration, we can proceed to discuss the importance of testing in this process, ensuring that the bot we develop works well in various conditions.

11.1.2 The Importance of Testing in Webhook Integration

Testing is a critical phase in the webhook integration process. Without adequate testing, you may encounter issues that are difficult to detect, such as unresponsive webhooks or data processing errors.

Testing helps ensure that your webhook functions properly and can handle various types of updates from the platform.

Testing in webhook integration is a crucial step to ensure that the bot works as expected and interacts efficiently with users.

Without proper testing, various problems may arise, ranging from communication errors to failures in responding to user commands. Therefore, it is important for developers to carry out a series of tests to ensure the reliability and optimal performance of the bot.

There are several reasons why webhook testing is essential:

1. ***Validating Webhook Functionality***: Testing ensures that the webhook you've configured actually receives and processes updates from the platform. This is crucial to make sure your bot works well in real-life situations.

2. ***Error Detection***: Testing allows you to find and fix errors in the code handling the webhook. These could be errors in data processing, connection issues, or configuration mistakes.
3. ***Reliability and Stability***: By conducting tests, you can ensure that your webhook is stable and reliable. This helps prevent disruptions to your bot service and ensures a smooth user experience.

To test webhooks, you can use various tools and techniques. A common method is using tools like Postman to send HTTP POST requests to your webhook URL and check the response received. Additionally, the platform where your bot operates often provides tools or features to directly check and test webhooks.

There are several types of tests needed in the webhook integration process. First, unit tests are used to test individual functions in the code separately. For example, we can test a function that handles updates from the platform, ensuring that it processes the data correctly. In `tests.py`, we can write the following test:

```
# File: apps/webhook/tests.py

from django.test import TestCase
from .views import handle_update  # Import the
function to be tested

class WebhookTests(TestCase):
    def test_handle_update(self):
        # Simulating an update received
        update = {
            'message': {
                'chat': {'id': 123},
                'text': '/start'
            }
        }
        response = handle_update(update)
        self.assertEqual(response, "Welcome!")  #
Ensure the output is correct
```

In addition to unit tests, integration tests are also important to ensure that different components of the system work well together. In this case, we can test the entire communication flow between the bot and the platform. A situation that can be tested is when a user sends a message to the bot, whether the bot can receive the message and send the correct response.

If testing is neglected, various situations can arise. For example, the bot may not respond correctly when a user sends a specific command, or it may fail to operate altogether if an error occurs in the webhook integration. This could lead to a poor user experience, potentially diminishing users' trust in the application we are developing. For instance, if a user sends a message to the bot and the bot cannot process the command due to a webhook setup error, such issues might go undetected until the application is in production, causing frustration for users.

Therefore, testing is not just an additional step but an integral part of the development process that helps ensure that the bot and webhook are properly integrated. By conducting comprehensive tests, we can prevent issues before they arise and improve the quality of the final product.

By understanding the concept of webhook integration and the importance of testing, you can ensure that your bot functions well and interacts with users in real-time. Next, we will discuss how to set up webhook integration for your bot on the platform.

11.2 Setting Up Webhook Integration For Bots

In the following steps, we will discuss the process of integrating the bot with a webhook in detail. This integration enables the bot to receive and process user messages in real-time, making it an effective and responsive communication tool. Before starting the integration process, there are some preparations needed to ensure everything runs smoothly.

First, make sure that your Django project is properly configured. This includes setting up the `settings.py` file, where we need to add the webhook URL that will be used by the bot. For example, we can add the following URL:

```
# File: settings.py

WEBHOOK_URL = 'https://example.com/webhook/'  # 
Replace with the appropriate webhook URL
```

After ensuring the basic configuration is correct, the next step is to set up a view that will handle updates from the bot. This view will receive data from the webhook and process it according to the defined logic. Here, we can create the following view in `views.py`:

```
# File: apps/webhook/views.py

@csrf_exempt
def telegram_webhook(request):
    # ...

@csrf_exempt
def whatsapp_webhook(request):
    # ...
```

With the view ready, we also need to add URL routing to direct requests to the appropriate view. This can be done by adding a new entry in `urls.py`:

```
# File: apps/webhook/urls.py

from django.urls import path
from .views import telegram_webhook, whatsapp_webhook

urlpatterns = [
    # Webhook endpoint for Telegram
    path('telegram/', telegram_webhook,
name='telegram_webhook'),
    # Webhook endpoint for WhatsApp
    path('whatsapp/', whatsapp_webhook,
name='whatsapp_webhook'),
]
```

Once these steps are complete, we need to configure the bot on the platform we are using, such as Telegram or WhatsApp, to use the webhook URL we created. At this stage, ensure the bot token obtained from the platform is correctly configured in our application.

With all the above configurations, the integration between the bot and the webhook is now ready to be tested. It is important to conduct tests to ensure that all components function properly and can communicate with each other. During this process, we can identify potential issues that may arise, allowing us to fix them before the application is released to users.

Integrating the bot with the webhook is not only a technical step but also lays a strong foundation for a better and more responsive user experience. By following these steps, we will be able to create a reliable and efficient system that meets users' needs in real time.

11.3 Implementing Webhook And Bot Integration

After setting up the webhook integration, the next step is to practically integrate the webhook with the bot.

At this stage, we will modify the view in `apps/bots/views.py` and the template `create_bot.html` so that the bot creation functionality can be integrated with the webhook. The updated code allows users to create bots and automatically set the appropriate webhook based on the selected platform.

11.3.1 Integration in `apps/bots` View

Now, update the view in `apps/bots/views.py` to use the handler that has been created. We need to ensure that the webhook view calls the appropriate handler based on the platform sending the data.

Here is the updated code:

```
# File: apps/bots/views.py

from django.shortcuts import render, redirect,
get_object_or_404
from django.contrib.auth.decorators import
login_required
from django.contrib import messages  # Added import
to use messages
from .forms import BotForm
from .models import Bot
from django.conf import settings
from apps.webhook.views import set_telegram_webhook,
set_whatsapp_webhook

WEBHOOK_URL = settings.WEBHOOK_URL

@login_required
def create_bot(request):
    if request.method == 'POST':
        form = BotForm(request.POST)
        if form.is_valid():
            bot = form.save(commit=False)
            bot.user = request.user
            bot.is_active = True  # Set bot as active
by default

            # Set webhook URL according to the
platform
            if bot.bot_type == 'telegram':
                bot.webhook_url =
f'{WEBHOOK_URL}/telegram/'
            elif bot.bot_type == 'whatsapp':
                bot.webhook_url =
f'{WEBHOOK_URL}/whatsapp/'

            bot.save()

            # Set webhook based on bot type
            if bot.bot_type == 'telegram':

set_telegram_webhook(bot.token_telegram,
bot.webhook_url)
                # Add success message for Telegram
bot
                messages.success(request, 'Telegram
bot successfully created!')
            elif bot.bot_type == 'whatsapp':
                set_whatsapp_webhook(bot.webhook_url)
                # Add success message for WhatsApp
bot
                messages.success(request, f'WhatsApp
bot created successfully. Please set the following
webhook URL in the Twilio dashboard:
{bot.webhook_url}')

            return redirect('manage_bots')
    else:
        form = BotForm()

    return render(request, 'bots/create_bot.html',
{'form': form})

@login_required  # Ensure that only logged-in users
can access this view
def manage_bots(request):
    bots = Bot.objects.filter(user=request.user)  #
Retrieve list of bots owned by the logged-in user

    if request.method == 'POST':
        if 'delete' in request.POST:  # Check if
there is a request to delete a bot
```

```
            bot_id = request.POST.get('delete')  #
Get the bot ID to be deleted from the form
            bot = get_object_or_404(Bot, id=bot_id)
# Find the bot by ID, or return 404 if not found
            bot.delete()  # Delete the bot from the
database
            messages.success(request, 'Bot
successfully deleted.')  # Add a message after the
bot is deleted
            return redirect('manage_bots')  # After
deleting the bot, return to the bot management page

    return render(request, 'bots/manage_bots.html',
{'bots': bots})  # Render the page with a list of
bots owned by the user
```

Explanation of Updated Code

1. ***Required Imports:*** This code imports `set_telegram_webhook` and `set_whatsapp_webhook` from the `apps.webhook.views` module. These functions will handle webhook settings based on the selected platform.
2. ***Setting Webhook URL:*** In the updated code, we assign `bot.webhook_url` using the configuration from `settings.py`. This ensures that each created bot has the correct and integrated webhook URL.
3. ***Webhook Integration:*** After saving the bot, we check the selected bot type. If the bot is of type Telegram, we call the `set_telegram_webhook` function with the token and webhook URL. Conversely, if the bot is of type WhatsApp, we retrieve the form data (Account SID, Auth Token, and WhatsApp number) and call `set_whatsapp_webhook`.

4. ***Redirect to Bot Management Page:*** After the save and webhook integration process is successful, the user will be redirected back to the bot management page using `redirect('manage_bots')`.

Adding `WEBHOOK_URL`

Add the webhook URL to `settings.py`. This will be used in the code to configure the webhook endpoint with the bot platform:

```
# File: platform_bot/settings.py

WEBHOOK_URL = 'https://ngrok-random_id.app'  # 
Replace with the obtained ngrok URL
```

With these improvements, the bot creation view is now effectively connected to the webhook system, enabling better interaction between users and the bots they create.

11.3.2 Fixing the Manage Bot Page

Here is the appropriate modification for `manage_bots.html`, where we will display a success message for a bot that has been successfully created.

```
<!-- File: apps/templates/bots/manage_bots.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
  <div class="container">
    <h1>Manage Bots</h1>

    <!-- Display success message if any -->
    {% if messages %}
      <div class="alert alert-success alert-
dismissible fade show" role="alert">
        {% for message in messages %}
          {{ message }}
        {% endfor %}
        <button type="button" class="btn-close" data-
bs-dismiss="alert" aria-label="Close"></button>
      </div>
    {% endif %}

    <table class="table table-striped">
      <thead>
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>Platform</th>
          <th>Token</th>
          <th>Status</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        {% for bot in bots %}
          <tr>
            <td>{{ bot.id }}</td>
            <td>{{ bot.name }}</td>
            <td>{{ bot.bot_type }}</td>
            <td>{{ bot.token_telegram }}</td>
            <td>{{ bot.is_active|
yesno:"Active,Inactive" }}</td>
            <td>
              <a href="" class="btn btn-
primary">Edit</a>
              <form method="post"
style="display:inline;">
                {% csrf_token %}
                <button type="submit" name="delete"
value="{{ bot.id }}" onclick="return confirm('Are you
sure you want to delete this bot?');" class="btn btn-
danger">Delete</button>
              </form>
            </td>
          </tr>
        {% empty %}
          <tr>
            <td colspan="6">No bots available.</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>

    <a href="{% url 'create_bot' %}" class="btn btn-
primary">Create New Bot</a>
  </div>
{% endblock %}
```

Explanation:

1. ***Success Message***: The success message (`messages.success`) will be displayed after a bot is successfully created (whether it's for Telegram or WhatsApp). This message will appear at the top of the `manage_bots.html` page.
2. ***Deleting Bots***: The success message is also displayed when a bot is successfully deleted, using `messages.success`.
3. ***Bootstrap Alert***: The success message uses Bootstrap's alert component with a dismissible feature, allowing users to close the message.

With this implementation, users will be notified whenever they successfully create or delete a bot, whether for Telegram or WhatsApp.

11.4 Testing Bot Creation

Testing is a crucial part of webhook and bot integration to ensure that all functions operate as expected. Here is a step-by-step guide for testing bot integration with webhooks, starting from running the Django server to using the bot on platforms like Telegram.

11.4.1 Testing Telegram Bot Creation

Testing Telegram bot creation involves several key steps, from obtaining the bot token to sending a message to verify that the integration has been successful. Below are the steps to follow when testing a newly created Telegram bot.

Obtaining a Telegram Bot Token

The first step is to obtain a token for the Telegram bot. Open the Telegram app and search for the user named ***BotFather***. BotFather is the official bot used to create new bots. Send the command `/newbot` to BotFather. BotFather will ask you to provide a name and username for the bot you want to create. Once completed, BotFather will provide an API token that you need to save, as it will be used in the Django application.

Running Ngrok and Configuring Settings

To test the webhook locally, we need to use ***Ngrok***. Ngrok allows us to create a tunnel from localhost to a publicly accessible URL. After installing Ngrok, run the following command in the terminal:

```
$ ngrok http 8000
```

This command will display a public URL that is directed to our local server. Note this URL, as it will be used to configure the Telegram bot webhook. Next, we need to set the environment variable in `settings.py` to include the Ngrok URL as the webhook. Ensure the `WEBHOOK_URL` is set as follows:

```
# File: settings.py
WEBHOOK_URL = 'https://<your-ngrok-url>'  # Replace
with the displayed Ngrok URL
```

Running the Django Server

Once the configuration is done, we need to run the Django server. In the terminal, navigate to the project directory and run the following command:

```
$ python manage.py runserver
```

This will start the Django server on `localhost:8000`, allowing us to access the application via the browser.

Accessing the Create Bot Page and Entering the Token

With the server running, open the browser and access the bot creation page at `http://localhost:8000/create_bot/`. Fill out the form with the bot name and the token obtained from BotFather. Also, ensure that you select Telegram as the bot type before submitting the form.

Sending a Test Message in Telegram

Once the bot is successfully created, we can try sending a message to the bot on Telegram. Find the newly created bot in the Telegram app by the name you provided. Send a message such as "/start" or "/help".

Successful Response in Ngrok

After sending the message, return to the terminal running Ngrok. Ngrok will display the logs of the requests received. If the integration is successful, we should be able to see logs indicating that the bot received the message and responded according to the webhook configuration.

```
Connections         ttl     opn     rt1     rt5
p50     p90
0.00    0.00    0.00    0.00

HTTP Requests
-------------
23:25:18.785 WIB POST /getpost/ 200 OK
```

If all steps have been followed correctly, we will receive confirmation that our Telegram bot has been successfully integrated and is working properly.

Thus, the testing of Telegram bot creation has been successfully completed.

11.4.2 Testing WhatsApp Bot Creation

In this section, we will focus on testing the creation of a WhatsApp bot using Twilio. This integration involves several crucial steps, from configuring the webhook to running the bot locally using the Django server. These steps are essential to ensure the bot can effectively receive and respond to messages from WhatsApp.

Configuring Webhook in Twilio

One of the key steps in creating a WhatsApp bot is configuring the webhook URL in Twilio so the bot can receive and respond to messages. A webhook is a mechanism that allows applications like Twilio to send data to a server you specify whenever there is related activity, such as receiving a message.

1. ***Access Twilio Console***: Once you have a Twilio account, log in to the Twilio Console. Here, you will find options to manage your WhatsApp project.
2. ***Configure Webhook***: Within the Twilio Console, open the project you created for the WhatsApp bot. Look for the *Messaging Services* or *WhatsApp* menu, where you will find the option to add a webhook. This webhook URL will receive messages from WhatsApp users and then process them in your Django application.
    - In the section *Webhook for incoming messages*, enter the webhook URL that will be used. For example, if you are using Ngrok to run the local server and expose it to a public URL, enter the following URL:

      ```
      arduino
      Copy code
      https://ngrok.random.id.app/whatsapp/
      ```

- Make sure to replace this with the URL generated by Ngrok or your domain if the app is already deployed.

3. ***Run Ngrok to Connect Webhook***: Since the Django server typically runs on localhost, you need to use Ngrok to provide public access to your local application. Run Ngrok using the command:

```
$ ngrok http 8000
```

After running, Ngrok will provide a public URL (e.g., `https://random.id.ngrok.io`). Use this URL as the webhook in Twilio, appending `/whatsapp/` to the path as shown in the example above.

Running the Django Server and Testing the Bot

Once the webhook is configured, run the Django server to ensure the bot can receive and process WhatsApp messages.

1. ***Run the Django server*** using the following command:

```
$ python manage.py runserver
```

2. ***Access the WhatsApp bot creation page*** through your browser at the following URL:

```
http://localhost:8000/create_bot/
```

Fill out the form with the necessary information to create a WhatsApp bot. However, since credentials like the *Account SID* and *Auth Token* are not required in this system, you only need to ensure the webhook URL is correctly configured in Twilio.

Sending Test Messages to the WhatsApp Bot

Once the bot is successfully created and the Django server is running, test the bot by sending a message to the WhatsApp number you registered with Twilio.

1. Open the WhatsApp app on your phone.
2. Send a message to the WhatsApp number connected to Twilio.
3. Check the logs in the terminal running Ngrok to see if the bot receives and processes the message.

If the webhook and server settings are correct, you will see logs in the Ngrok terminal indicating that the message was successfully received and processed by the bot.

By following these steps, you can test a WhatsApp bot integrated with Twilio and ensure the webhook works correctly. Always use an active Ngrok URL during local development, and thoroughly test to ensure the WhatsApp bot functions as expected.

By following the steps above, you will be able to integrate the bot with webhooks on platforms like Telegram and WhatsApp, and test that everything is working correctly. Make sure to always perform thorough testing on each platform and resolve any errors that arise through logs or the Django console.

Chapter Conclusion

In this section, we focused on the process of **integrating bots with webhooks**, which is a crucial element in connecting bots with platforms like Telegram and WhatsApp. Starting with an ***Introduction to Webhook Integration with Bots***, we understood how webhooks act as a communication bridge between bots and external platforms, as well as the importance of testing to ensure that the webhook functions properly.

In the ***Setting Up Webhook Integration for Bots*** section, we built the technical foundation for integrating the bot with webhooks on various platforms. This process included **structuring the webhook** to handle updates from multiple platforms simultaneously. In this section, we learned how to connect the webhook to the bot in Django, including setting up views and handlers to process updates from Telegram and WhatsApp. We also configured the necessary settings for each platform to ensure that the bot works optimally.

In the final part, ***Webhook and Bot Integration***, we completed the integration by creating functions to process updates from each platform. We updated the views in `apps/bots` to handle various types of messages and updates from external platforms. This section also emphasized the importance of testing, including the process of creating, editing, and managing bots on the platform, which was tested using the Django server and Ngrok to ensure that the webhook is connected and working correctly.

By completing this chapter, we not only learned how to integrate bots with webhooks but also ensured thorough testing at each step of the process. This chapter provided a deep technical understanding and practical skills in building, managing, and testing bot integrations with various platforms through webhooks, making your bot more responsive and efficient in handling user interactions.

Chapter 12 - Creating Features for Bots

In this chapter, we will discuss how to build essential features for managing bots on the Django platform, including adding commands, editing bots, and managing existing commands. These features are crucial because commands on a bot serve as the primary mechanism for interaction between users and the bot. By setting up the right commands, the bot can respond to user requests in an efficient and structured manner.

This chapter begins with an introduction to commands on bots and the importance of this feature in building strong interactions between users and the bot system. After that, we will explore the technical steps involved in setting up models, forms, and views to add and edit commands. Additionally, we will integrate these features with an activity log, ensuring that every significant action such as the creation, editing, or deletion of commands and bots is properly recorded.

Throughout this chapter, we will learn the technical processes from creating models and forms to integrating with the activity log model, as well as how each component interconnects to build an effective and efficient bot management system. At the end of the chapter, we will also review the complete code that constitutes all these features, enabling the developed bot to function smoothly on various platforms like Telegram and WhatsApp.

Thus, this chapter serves as an important foundation in the development of functional and flexible bots, ready to be utilized in various user interaction scenarios.

12.1 Introduction To Command Features On Bots

When building a bot, one of the most important and frequently used features is the **command**. A command on a bot is an instruction entered by the user to trigger a specific action. In bot platforms like Telegram, commands usually begin with the '/' character, such as /start or /help. This feature allows the bot to respond to users in a structured and logical manner.

12.1.1 What Are Commands on Bots?

Commands are instructions given by users to the bot to perform specific functions. In the context of bot applications, such as Telegram, commands can be used to direct the bot to display messages, initiate services, or execute other programmed actions. For example, the /start command is typically used to initiate the initial interaction with the bot and present basic information to the user.

Commands on bots can be understood as instructions provided by users to guide the bot in performing specific tasks. This is the primary way for users to interact with the bot, and it can be a simple text programmed to trigger various actions. For instance, in the context of a Telegram bot, users might use commands like /start to begin interaction or /help to obtain information about available functions.

Commands on bots function like shortcuts that allow users to interact without having to type lengthy texts. Each command is mapped to a specific function in the bot's backend, and when the command is received, the bot will execute the corresponding function.

The importance of commands in bot interaction cannot be underestimated. With commands, users can easily and quickly access special features provided by the bot. Commands provide structure to the communication, allowing users to understand how to interact with the bot in a more intuitive manner. Additionally, commands enable developers to handle various different usage scenarios, ranging from providing information to executing more complex tasks.

The process of adding and editing commands is also an important aspect of bot development. Developers must design a system that allows commands to be added, modified, or deleted as needed. With this system, bots can continue to evolve and adapt to user feedback and changing requirements over time.

In this chapter, we will delve deeper into how to add command features to our bot, including the technical steps and best practices to consider. We will begin by discussing the models needed for commands, then move on to the development of forms, views, and templates that support these interactions. With these steps, we will create a bot that is not only responsive but also easy to manage and further develop.

12.1.2 Importance of Commands in Bot Interaction

In the world of bot development, commands play a crucial role in shaping the interaction between bots and their users. Commands not only provide a way for users to communicate with bots but also determine how bots respond and perform specific tasks. By understanding the importance of commands, we can design more effective and efficient bots.

Commands are essential in creating an interactive and user-friendly experience. With commands, users can give instructions to the bot directly without confusion. For example, an e-commerce bot might have a command like `/order` to initiate the ordering process or `/status` to check the status of a previous order.

One of the main reasons commands are so important is that they provide a clear structure in interactions. Users do not have to guess how to use the bot; they simply enter the predefined command. For instance, when a user types `/weather`, they expect the bot to provide the current weather information. With commands in place, bots can deliver relevant responses quickly, enhancing the user experience.

Additionally, commands allow developers to better handle various scenarios. By defining different commands, developers can guide the bot to perform specific functions according to user needs. This creates flexibility in how the bot operates, enabling new features to be added easily without overhauling the entire system. For example, a developer can add a new command to access specific information or modify bot settings by simply adding a few lines of code.

Commands also serve as a tool to facilitate more effective communication. In situations where information needs to be conveyed quickly, such as in reminders or notifications, commands can be used to trigger automatic responses. This not only saves time but also ensures that users receive the information they need promptly.

In terms of security, commands can help control access to specific bot functions. By defining who can use certain commands, developers can protect sensitive data and ensure that only authorized users can execute critical commands.

With commands, users do not need to remember various complex texts or instructions; they only need to type the defined command, and the bot will process it immediately. This increases the efficiency of interactions between users and bots, helping to ensure that interactions proceed as designed.

By effectively understanding and implementing commands, we can enhance user interactions and ensure that the bot operates well, meets user expectations, and provides a satisfying experience. In the next chapter, we will further discuss how to add and edit commands in our bot so that it can function as intended.

12.1.3 The Process of Adding and Editing Commands

Adding and editing commands on a bot is an important process that allows developers to optimize user interactions. This process consists of several steps, starting from defining commands, implementing them in code, to ensuring that these commands can be edited as needed.

The first step in adding a command is to define it. This includes determining the command's name, the desired functionality, and how the bot will respond to that command. For example, if we want to add a command to get weather information, we need to decide on the command name, such as `/weather`, and determine what parameters may be needed, such as location or the type of weather information requested.

Once the command is defined, the next step is to implement it in code. Here, we will create a function that handles the command. For instance, we can use Django to create a view that responds when the command is received. The following code is a simple example for handling the weather command:

```python
# File: apps/bots/views.py
from django.http import JsonResponse
import requests

def weather_command(request):
    location = request.GET.get('location',
'default_location')
    # Fetching weather information from the API
    response =
requests.get(f'https://api.weatherapi.com/v1/current.
json?key=YOUR_API_KEY&q={location}')
    weather_data = response.json()

    return JsonResponse(weather_data)  # Returning
weather data in JSON format
```

In this code, we create a function `weather_command` that receives location as a parameter, accesses the weather API, and returns the weather data in JSON format. This way, we ensure that the added command can provide relevant and useful information to the user.

After the command is added, it is important to ensure that we also provide a way to edit the command in the future. This process typically involves creating a user interface (UI) that allows users to view, add, or edit existing commands. This can be done by using forms in Django, where users can input the desired changes and save them to the database.

A simple example of a form for editing a command might look like this:

```
# File: apps/bots/forms.py
from django import forms
from .models import Command

class CommandForm(forms.ModelForm):
    class Meta:
        model = Command
        fields = ['name', 'description']  # For
example, only name and description can be edited
```

Here, we use Django's ModelForm to create a form that allows users to edit the command's name and description. Once this form is ready, we need to prepare a view that can handle input from the form and save changes to the database.

By understanding the process of adding and editing commands, we not only expand the functionality of the bot but also provide flexibility for developers to customize the bot according to user needs. In the next section, we will further discuss how to set up features for editing bots and commands in more detail.

Each of these steps will be discussed in detail in the next chapter, from preparing models to testing the command features on the bot. This ensures that every command created functions correctly and according to the desired logic.

12.2 Setting Up Models And Forms

In bot development, models are a crucial foundation. Models define the data structure that we will use, including commands and responses that will be processed by the bot. In this subsection, we will create models and forms for the commands that will be used by the bot.

12.2.1 Creating a Model for Commands

We will use Django ORM (Object-Relational Mapping) to create the `BotCommand` model. This model will store information about commands related to a specific bot, including the command name and the response that should be provided when the command is received.

Here is the implementation of the model:

```
# File: apps/bots/models.py

from django.db import models

class BotCommand(models.Model):
    bot = models.ForeignKey(Bot,
related_name='commands', on_delete=models.CASCADE)
    command = models.CharField(max_length=255,
help_text="Enter commands like '/start', 'hello',
etc.")
    response = models.TextField(help_text="Enter the
response that will be sent when the command is
received.")

    def __str__(self):
        return f"Command: {self.command} -> Response:
{self.response}"

    class Meta:
        unique_together = ['bot', 'command']  #
Ensure each bot only has one command with the same
name.
        verbose_name = "Bot Command"
        verbose_name_plural = "Bot Commands"
```

Let’s discuss this model in more detail. First, we have a ForeignKey relationship that connects each command to a specific bot. This means that each command must be associated with one bot, and we use `related_name='commands'` to make it easier to access the commands related to a bot object.

The `command` field uses a `CharField` data type with a maximum length of 255 characters. This will be used to store the command name, for example, `/start` or `hello`. We also provide `help_text` that gives instructions on how to use this field, making it easier for developers or other users interacting with the form to understand how to fill in the data.

The `response` field uses a `TextField` to store the response that will be sent when the command is received. This allows us to store longer and more informative messages that can be conveyed to the user.

The `__str__` method is defined to provide an informative string representation of the model object. When we print a `BotCommand` object, we will see the command name and the associated response, which is very useful for debugging or when displaying data in the admin panel.

In the Meta section, we define two important things. First, we set `unique_together`, which ensures that each bot can only have one command with the same name. This prevents unwanted command duplication and maintains data consistency. Second, we use `verbose_name` and `verbose_name_plural` to provide better and more descriptive names in the Django admin panel.

This model provides a foundation for us to store command data in the database. After the model is created, don't forget to create and run migrations so that the `BotCommand` table can be created in the database:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Thus, we have completed the first step in setting up the command feature on the bot by creating a model that will store commands and corresponding responses. In the next subsection, we will proceed to create views and an interface to add new commands through the admin panel or a special page.

With this model, we are ready to store commands related to the bot in the database, and the next step is to create a form that allows users to add or edit those commands. In the following section, we will further discuss this process.

12.2.2 Creating Forms for Commands

After defining the `BotCommand` model, the next step is to create a form that will be used to add and edit commands. This form will provide an interface for users to enter the necessary data, such as the command name and the response that the bot will send. We will use Django’s `ModelForm` to create this form, so all validation and data management will be handled automatically. This form will be used to take input from the user (admin or authorized user) and then store that data in the database using the `BotCommand` model.

In a Django application, creating forms can be done easily using `ModelForm`, which will automatically generate a form based on the model we created earlier. Here, we will create a form for adding commands and responses.

The `BotCommandForm` allows users to enter commands and their responses simply while providing an easy-to-use interface with placeholders and help text.

Here is the implementation of the `BotCommandForm`:

```
# File: apps/bots/forms.py

from .models import BotCommand
from django import forms

class BotCommandForm(forms.ModelForm):
    class Meta:
        model = BotCommand
        fields = ['command', 'response']  # Fields to
be displayed in the form
        widgets = {
            'command':
forms.TextInput(attrs={'placeholder': 'Example:
/start, /help'}),
            'response': forms.Textarea(attrs={'rows':
3, 'placeholder': 'Enter the response to be sent.'}),
        }
        help_texts = {
            'command': 'Enter commands like "/start",
"/help", or others.',
            'response': 'Enter the response that will
be sent when the command is received.',
        }
```

Let’s break down the components of this form. First, we import `forms` from Django and the `BotCommand` model that we created earlier. Then, we define `BotCommandForm` as a subclass of `forms.ModelForm`, which allows us to automatically link this form to the existing model.

In the Meta class, we specify the model to be used, which is `BotCommand`, and determine the fields to be displayed in the form. In this case, we choose `command` and `response`. This ensures that the form only requests relevant information from the user.

Next, we use widgets to configure the form's appearance. For the `command` field, we use `TextInput` with a placeholder attribute that provides examples of commands that can be entered. This helps users understand the expected format. For the `response` field, we use `Textarea` with a specified number of rows, as well as a placeholder to provide further instructions regarding the content of the response.

Additionally, we also provide `help_texts`, which serve to give clearer instructions regarding each field. This will appear below each input in the form, providing direct guidance to users on what they should enter.

With this form ready, users can easily add new commands or edit existing ones through a more intuitive interface. Next, we will discuss how to create a view that will handle the logic of saving and editing commands through this form.

After creating the form, the next step is to integrate it into the view. This form will be used to display the command addition and editing page.

In the next section, we will discuss how to integrate this form into the view and ensure that user input can be processed and stored correctly in the database.

12.3 Setting Up The Edit Bot Feature

In this section, we will set up the edit bot feature, starting with preparing views, the edit bot template, and URL routing for editing the bot. On the edit bot page, users can edit the bots they have created.

12.3.1 Creating a View for Editing the Bot

After preparing the necessary models and forms, the next step is to create a view to manage the bot editing process. This view is responsible for retrieving the bot data to be edited, loading the form with the existing data, and processing the storage of data after the user makes changes.

Let’s start by defining the view for editing the bot. Here’s an example implementation of the `edit_bot` view:

```
# File: apps/bots/views.py

from .forms import BotForm  # Ensure BotForm is
defined in forms.py

def edit_bot(request, bot_id):
    bot = get_object_or_404(Bot, id=bot_id)  #
Retrieve the bot based on ID
    if request.method == 'POST':
        form = BotForm(request.POST, instance=bot)  #
Load data into the form
        if form.is_valid():
            form.save()  # Save the edited data
            return redirect('manage_bots')  #
Redirect to the bot management page
    else:
        form = BotForm(instance=bot)  # If not POST,
create form with existing bot data

    context = {
        'form': form,
        'bot': bot,
    }
    return render(request, 'bots/edit_bot.html',
context)  # Render the template with context
```

Here, we start by importing the necessary functions from Django, including `render`, `get_object_or_404`, and `redirect`. Then, we retrieve the `Bot` model and the `BotForm` we defined earlier.

The `edit_bot` function receives the request and `bot_id` as parameters. First, we retrieve the corresponding bot object using `get_object_or_404`. If the bot is not found, the user will be redirected to a 404 error page.

Next, we check if the request method is POST. If it is, this means the user has submitted the form. We then create an instance of `BotForm` with the received data and link it to the existing bot object. If the form is valid, we save the changes and redirect the user to the bot management page using `redirect`.

If the request method is not POST, we create an instance of `BotForm` with the data from the existing bot object. This will fill the form with the existing information, allowing the user to view and edit the bot’s details.

Finally, we prepare the context containing the form and the bot object to be passed to the `edit_bot.html` template. This template will be used to display the form to the user, allowing them to edit the desired bot information.

With this view ready, users can now easily edit existing bots. In the next stage, we will discuss how to create a template to display this edit form.

12.3.2*1* Creating a Template for Editing the Bot

After preparing the view for editing the bot, the next step is to create a template that will display the edit form to the user. This template will provide an intuitive interface for changing the bot's information, allowing users to easily modify it according to their needs.

Let’s create the `edit_bot.html` template. Here’s an example:

```
<!-- File: apps/templates/bots/edit_bot.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <div class="card">
        <div class="card-header">
            <h5 class="card-title">Edit Bot:
{{ bot.name }}</h5>
        </div>
        <div class="card-body">
            <form method="post">
                {% csrf_token %}
                {{ form.as_p }}  <!-- Display all
form fields as paragraphs -->

                <button type="submit" class="btn btn-
primary">
                    Save Changes
                </button>
                <a href="{% url 'manage_bots' %}"
class="btn btn-secondary">Back</a>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

In this template, we start by inheriting from the base dashboard template (`dashboard/base.html`), which provides the overall structure for the page. We then open the content block to fill in the relevant section.

Inside the content block, we create a Bootstrap container to hold a card that contains the edit bot form. The card's title displays the name of the bot being edited, providing clear context to the user.

The form uses the POST method to send data back to the server. We include a CSRF token for security, which Django requires to protect against CSRF attacks.

To display the form fields, we use `{{ form.as_p }}`, which automatically generates HTML for all fields in the form, displayed as paragraphs. This simplifies the layout and maintains consistent styling.

After the form fields, we provide two buttons: one to save changes and the other to return to the bot management page. The save button uses the Bootstrap `btn btn-primary` class to give it a prominent appearance, while the back button uses the `btn btn-secondary` class to indicate an alternative action.

With this template, users will have a clean and functional interface to edit the bot's information. Next, we will discuss how to set up URL routing to connect this edit bot view.

12.3.3 Setting Up URL Routing for the Bot Edit Feature

After we have prepared the view and template for the bot edit feature, the next step is to set up URL routing that will connect user requests to the appropriate view. This setup is crucial to ensure that users can access the bot edit page using the corresponding URL.

To configure URL routing, we will add a new path in the `urls.py` file found in the bot application. Here is an example code that we can use:

```
# File: apps/bots/urls.py
from django.urls import path
from . import views

urlpatterns = [
    # URL for the bot edit page
    path('edit_bot/<int:bot_id>/', views.edit_bot,
name='edit_bot'),  # Using bot_id as a parameter
]
```

In the code above, we use Django's `path()` function to define a new URL. This URL is designed to handle requests to the bot edit page. We add the parameter `<int:bot_id>` in the URL, which means we will receive the bot ID as an integer. This parameter will be passed to the `edit_bot` view that we created earlier, allowing us to identify the bot to be edited.

By naming this URL `'edit_bot'`, we also make it easier to refer to this URL in templates and other code. This is particularly useful when we want to create links to the bot edit page, such as in the back button we added previously.

After adding this path, make sure to test it to ensure that the routing works correctly. With this step, we have connected the user interface for editing the bot with the relevant functionality in our application.

With the URL routing set up, we are ready to move on to the next step, which is preparing the features for the bot command.

12.4 Preparing Features To Add Bot Command

In this section, we will set up the feature to add commands, starting from preparing views, the template for adding commands, and URL routing for editing the bot. On the add command page, users can create commands and responses for their bots.

12.4.1 Creating a View to Add Command

After successfully creating the form for commands, the next step is to create a view that will process this form. This view is responsible for displaying the page for adding a new command, receiving user input, and saving data to the database.

At this stage, we will create a function-based view that handles the process of adding commands for an existing bot. Users who want to add a command must be logged in, and only the bot's owner can add commands for their bot.

Here is the complete code for the `add_command` view:

```
# File: apps/bots/views.py

from .models import BotCommand
from .forms import BotCommandForm
```

```
@login_required
def add_command(request, bot_id):
    # Get the bot based on ID, ensure the bot belongs
to the logged-in user
    bot = get_object_or_404(Bot, id=bot_id,
user=request.user)

    # If the request method is POST, process the form
    if request.method == 'POST':
        form = BotCommandForm(request.POST)
        if form.is_valid():
            command = form.save(commit=False)  #
Don't save to the database yet, assign the bot first
            command.bot = bot  # Assign the bot to
the command
            command.save()  # Save the command to the
database
            return redirect('manage_bots',
bot_id=bot.id)  # Redirect to the bot edit page after
successfully adding the command
    else:
        form = BotCommandForm()  # If GET, display an
empty form

    # Render the add_command.html template with the
form and bot data
    return render(request, 'bots/add_command.html',
{'form': form, 'bot': bot})
```

The `add_command` function begins by checking whether the user is logged in using the `@login_required` decorator.

Next, we retrieve the bot object based on the provided `bot_id`. Using `get_object_or_404`, we ensure that the requested bot exists and is owned by the logged-in user. If the bot is not found, the user will be directed to a 404 page.

If the request method is POST, we initialize the form with the received data. After validating the form, we save the newly added command, but do not save it directly to the database yet. As a first step, we link the command to the corresponding bot, and then we save the command to the database.

After the command is successfully added, the user will be redirected back to the bot edit page using `redirect`. If the request method is GET, we display an empty form for the user to fill in the new command data.

Finally, we render the `add_command.html` template, providing the form and bot data to be displayed to the user. With this view, we have laid the foundation needed to add a new command to the bot.

After creating the view, the next step is to create the HTML template to display this form. This template will serve as the interface for users to add new commands. In the next sub-section, we will discuss the creation of this HTML template.

12.4.2 Creating a Template to Add Command

After setting up the view for adding commands, the next step is to create the HTML template that will be used to display the command addition form to users. This template is the interface that users will use to enter information about the new command they wish to add to their bot.

Here is the code for the `add_command.html` template:

```
<!-- File: apps/templates/bots/add_command.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <h2>Add Command for Bot: {{ bot.name }}</h2>

    <div class="card">
        <div class="card-header">
            <h5 class="card-title">Add Command</h5>
        </div>
        <div class="card-body">
            <form method="post">
                {% csrf_token %}

                <!-- Form Group for Command Field -->
                <div class="form-group">
                    <label
for="{{ form.command.id_for_label }}">Command</label>
                    <input type="text" name="command"
id="{{ form.command.id_for_label }}"
value="{{ form.command.value }}" class="form-
control">
                </div>

                <!-- Form Group for Response Field --
>
                <div class="form-group">
                    <label
for="{{ form.response.id_for_label }}">Respons
e</label>
                    <textarea name="response"
id="{{ form.response.id_for_label }}" class="form-
control">{{ form.response.value }}</textarea>
                </div>

                <button type="submit" class="btn btn-
primary">Add Command</button>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

In this template, we use Django's templating syntax to set up the structure of the page. This template extends the base template we use in the application, referred to as `base.html`. By using the `content` block, we place specific content for this page.

Within the content, we display the page title showing the name of the bot for which the command is being added. Next, we create a card containing the form for adding a command. This form uses the POST method to send data to the server.

Inside the form, we use a CSRF token to protect against CSRF attacks. Then, we create two form groups: one for the command field and another for the response field. The command field is filled with a text input, while the response field is filled with a textarea to allow users to enter longer responses.

Finally, there is a button to submit the form labeled "Add Command." After the user fills out the form and clicks this button, the data will be processed by the view we created earlier. This template is designed to provide an intuitive user experience for adding commands to their bots.

With this template, you now have a user interface ready to be used for adding commands to the bot. The next step is to add URL routing and settings to integrate this template with other features in the application, which will be discussed in the next sub-section.

12.4.3 Setting Up URL Routing for the Add Command Feature

To ensure that the add command feature can be accessed by users, we need to set up URL routing in Django. URL routing connects the views we have created with URLs that can be accessed via a browser. This process involves adding a new route to the application's URL configuration and ensuring that the `add_command` view can be accessed through the appropriate URL.

The URL configuration for a Django application is usually found in the `urls.py` file. In this project, we need to add a new route in the `urls.py` file located in the `bots` application directory.

Here is an example code to set up URL routing:

```
# File: apps/bots/urls.py

from django.urls import path
from .views import add_command

urlpatterns = [
    # Other routes here...
    path('add_command/<int:bot_id>/', add_command,
name='add_command'),
]
```

In the code above, we import the `add_command` function from `views.py`. Then, we add a new entry in `urlpatterns`, which defines the route for adding a command. This route uses the path `add_command/<int:bot_id>/`, where `<int:bot_id>` is a parameter that will capture the ID of the bot for which the command is being added.

In this way, when a user accesses a URL like `yourdomain.com/add_command/1/`, our application will call the `add_command` function, and the relevant bot ID will be processed to add the new command. This route is named 'add_command,' allowing us to easily reference it in other templates and views.

Proper URL setup is essential to keep our application structure organized and easy to navigate. Once routing is set up, users can easily add commands for their bots by following the specified URLs.

12.4.4 Displaying the Command List

After we successfully added commands to the bot, the next step is to display the list of these commands on the bot management page. This way, users can see all the commands they have created and make edits or deletions if needed.

To display the list of commands, we will modify the `manage_bots.html` template to add a section that shows the commands related to each bot. Here's how we can do it:

```
<!-- File: apps/templates/bots/manage_bots.html -->

{% extends 'dashboard/base.html' %}
{% block content %}
  <div class="container">
    <h1>Manage Bots</h1>

    <!-- Display success message if available -->
    {% if messages %}
      <div class="alert alert-success alert-
dismissible fade show" role="alert">
        {% for message in messages %}
          {{ message }}
        {% endfor %}
        <button type="button" class="btn-close" data-
bs-dismiss="alert" aria-label="Close"></button>
      </div>
    {% endif %}

    <table class="table table-striped">
      <thead>
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>Platform</th>
          <th>Token</th>
          <th>Status</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        {% for bot in bots %}
          <tr>
            <td>{{ bot.id }}</td>
            <td>{{ bot.name }}</td>
            <td>{{ bot.bot_type }}</td>
            <td>{{ bot.token_telegram }}</td>
            <td>{{ bot.is_active|
yesno:"Active,Inactive" }}</td>
            <td>
              <a href="" class="btn btn-
primary">Edit</a>
              <form method="post"
style="display:inline;">
                {% csrf_token %}
                <button type="submit" name="delete"
value="{{ bot.id }}" onclick="return confirm('Are you
sure you want to delete this bot?');" class="btn btn-
danger">Delete</button>
              </form>
            </td>
            <td>
              <!-- Display related commands for each
bot -->
              <ul>
                {% for command in bot.commands.all %}
                  <li>{{ command.command }} -
{{ command.response }}</li>
                {% empty %}
                  <li>No commands available.</li>
                {% endfor %}
              </ul>
              <a href="{% url 'add_command' bot.id
%}" class="btn btn-success">Add Command</a>
            </td>
          </tr>
        {% empty %}
          <tr>
            <td colspan="6">No bots available.</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>

    <a href="{% url 'create_bot' %}" class="btn btn-
primary">Create New Bot</a>
  </div>
{% endblock %}
```

In the code above, we added a new column to the table that displays commands for each bot. By using `{% for command in bot.commands.all %}`, we can iterate over all commands related to that bot. If no commands are available, the message "No commands available." will be displayed.

Additionally, we provide a button to add new commands that links to the add command page we created earlier. This gives users quick access to enrich their bot's interactions by adding more commands.

12.4.5 Improving the Update Handler

To ensure that the bot can respond according to the added commands, we need to improve the handler for the bot to allow it to respond with the added commands.

Here’s how to fix the handler so that the bot can respond correctly:

Telegram.py

```
# File: apps/bots/services/telegram.py
from apps.bots.models import Bot, BotCommand
import requests

def handle_update(update):
    chat_id = update['message']['chat']['id']
    text = update['message']['text'].lower()  # 
Convert to lowercase for consistency in comparison

    bot = 
Bot.objects.filter(bot_type='telegram').first()  # 
Assuming we take the first bot, can be adjusted

    if bot:
        # Check if the command matches the one in the 
BotCommand model
        command = BotCommand.objects.filter(bot=bot, 
command__iexact=text).first()

        if command:
            response_text = command.response
        else:
            # If the command is not found, use
default commands /start and /help
            if text == "/start":
                response_text = bot.start_message
            elif text == "/help":
                response_text = bot.help_message
            else:
                response_text = 'Unknown command.
Type /help to see the list of commands.'

        send_message(bot.token, "sendMessage", {
            'chat_id': chat_id,
            'text': response_text
        })

def send_message(token, method, data):
    telegram_api_url =
f'https://api.telegram.org/bot{token}/{method}'
    response = requests.post(telegram_api_url,
data=data)
    if response.status_code != 200:
        logger.error(f"Failed to send message:
{response.text}")  # Log error if failed
    return response
```

By linking the handler to the BotCommand model, the bot can now respond according to the commands that have been added.

Whatsapp.py

```
# File: apps/bots/services/whatsapp.py

from django.http import HttpResponse  # For returning
HTTP response
from django.views.decorators.csrf import csrf_exempt
# To allow view access without CSRF validation
```

```
from twilio.twiml.messaging_response import
MessagingResponse  # For creating responses to Twilio
API
from apps.bots.models import Bot, BotCommand  #
Importing Bot and BotCommand models from bots app
import logging  # For logging activities and errors

logger = logging.getLogger(__name__)  # Initialize
logger

@csrf_exempt  # Decorator to disable CSRF validation
on this view
def handle_update(update):
    # Get the sender and message body
    user = update['From']
    message = update['Body'].strip().lower()  #
Convert message to lowercase for comparison

    # Logging the received message
    logger.info(f'{user} says {message}')

    # Retrieve the first WhatsApp bot from the
database
    bot =
Bot.objects.filter(bot_type='whatsapp').first()
    response = MessagingResponse()  # Create response
using Twilio's MessagingResponse

    if bot:  # If bot is found
        # Check if the command matches the one in the
BotCommand model
        command = BotCommand.objects.filter(bot=bot,
command__iexact=message).first()

        if command:
            response_text = command.response  # Get
the response from the command
        else:
            # If the command is not found, use
default commands /start and /help
            if message == "/start":
                response_text = bot.start_message  #
Welcome message
```

```
            elif message == "/help":
                response_text = bot.help_message  # 
Help message
            else:
                response_text = 'Unknown command. 
Type /help to see the list of commands.'  # Default 
message

        response.message(response_text)  # Send 
response message
    else:
        response.message('Bot not found.')  # If bot 
is not found in the database

    # Return response as HTTP response
    return HttpResponse(str(response))
```

Explanation of Changes:

1. ***Importing the BotCommand Model:*** We import the BotCommand model to retrieve the user-defined commands from the database.
2. ***Command Check:*** Similar to the Telegram code, we now check if the received message matches any commands in the BotCommand model. If a command is found, the response is taken from `command.response`.
3. ***Default Messages:*** If no commands match those in the database, we use default commands like `/start` and `/help`.
4. ***Response:*** The response from Twilio is generated using `MessagingResponse`, and if the bot is not found, an appropriate message is sent back.

With this approach, the handling for WhatsApp becomes more consistent with the handling for Telegram, making it easier for maintenance and further development.

With this feature, bot management becomes more intuitive and organized, giving users the ability to manage their commands more effectively.

12.5 Setting Up The Feature To Edit Commands

After setting up the feature to add commands to the bot, the next step is to create a feature to edit existing commands. Editing commands allows users to update the commands and responses sent by the bot. This process involves creating a view that enables users to edit commands through a web interface.

12.5.1 Creating a View to Edit Commands

Once we have successfully added the feature to add commands, it is now time to give users the ability to edit existing commands. This feature is important so that users can update commands and responses according to their needs. In this subsection, we will create a view that handles the command editing process.

In the first step, we need to define the `edit_command` function in the `views.py` file. This function will accept `command_id` as a parameter to identify the command to be edited. If the command is found, we will display an edit form with the existing data.

Here is the implementation of the function:

```
# File: apps/bots/views.py

@login_required
def edit_command(request, command_id):
    # Retrieve command based on ID
    command = get_object_or_404(BotCommand,
id=command_id)  # Fetch the command; if not found, it
returns 404
    bot = command.bot  # Retrieve the bot associated
with the command

    if request.method == 'POST':
        # If POST method, process the form data
        form = BotCommandForm(request.POST,
instance=command)  # Use the command instance to
populate the form
        if form.is_valid():
            form.save()  # Save the command changes
            return redirect('manage_bots',
bot_id=bot.id)  # Redirect to the bot edit page after
saving
    else:
        # If GET method, display the form with the
existing command data
        form = BotCommandForm(instance=command)  #
Show the form with existing data

    return render(request, 'bots/edit_command.html',
{'form': form, 'bot': bot})  # Render the template
with the form and bot data
```

In the above function, we use the `@login_required` decorator to ensure that only logged-in users can access this feature. First, we attempt to retrieve the command based on the received `command_id`. If the command is not found, the user will get a 404 page. After that, we retrieve the bot associated with the command so that we can redirect the user back to the bot edit page after the editing is completed.

If the received request method is POST, this means the user has filled out the form and submitted the data. We will create an instance of `BotCommandForm` with the submitted data and associate it with the command being edited. If the form is valid, we save the changes and redirect to the appropriate bot edit page. Conversely, if the request method is GET, we only display the edit form with the existing command data.

With this editing feature, users can easily adjust commands according to their bot's needs, enhancing the flexibility and ability of the bot to provide more relevant responses.

12.5.2 Creating a Template for Editing Commands

Once we have a functioning view for editing commands, the next step is to create a template that will display the edit form. This template will provide an intuitive user interface for updating existing commands.

We will create a file named `edit_command.html` in the `templates/bots` folder. This template will inherit from the existing base template `base.html`, maintaining consistency with the application’s appearance. Here is the implementation of the template:

```
<!-- File: apps/templates/bots/edit_command.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <h2>Edit Command for Bot: {{ bot.name }}</h2>

    <div class="card">
        <div class="card-header">
            <h5 class="card-title">Edit Command</h5>
        </div>
        <div class="card-body">
            <form method="post">
                {% csrf_token %}

                <!-- Form Group for Command Field -->
                <div class="form-group">
                    <label
for="{{ form.command.id_for_label }}">Command</label>
                    <input type="text" name="command"
id="{{ form.command.id_for_label }}"
value="{{ form.command.value }}" class="form-
control">
                </div>

                <!-- Form Group for Response Field --
>
                <div class="form-group">
                    <label
for="{{ form.response.id_for_label }}">Respons
e</label>
                    <textarea name="response"
id="{{ form.response.id_for_label }}" class="form-
control">{{ form.response.value }}</textarea>
                </div>

                <button type="submit" class="btn btn-
primary">Update Command</button>
            </form>
        </div>
    </div>
```

```
</div>
{% endblock %}
```

In the above template, we utilize the `content` block to define the page content. The page title reflects the bot being edited, providing clear context to the user. The edit form is built using the POST method to send data to the server.

Each field in the form is represented by the appropriate HTML elements. We display fields for the command and response, using existing values so that users can see and edit the current data. Labels for each field are also included to enhance accessibility and user understanding.

After users fill out the form and press the "Update Command" button, the updated data will be sent to the `edit_command` view for processing. If everything goes well, the user will be redirected back to the relevant bot edit page, where the changes can be seen and verified.

With this template, users gain an easy and efficient experience in editing their commands, allowing for flexibility in bot configurations.

With the HTML template ready, users can easily edit commands through the web interface. The next step is to ensure that all features related to editing commands work properly and verify that the updated data is correctly saved in the database.

12.5.3 Setting Up URL Routing for the Edit Command Feature

After setting up the view and template for editing commands, the next step is to set up the URL routing. This routing is important to connect the URL entered by the user with the appropriate view function. With the correct setup, users can easily access the edit command page.

We will add a new route in the `urls.py` file within our bots application. Here is the code snippet to be added:

```
# File: apps/bots/urls.py

from django.urls import path
from . import views

urlpatterns = [
    # Other URLs
    path('edit_command/<int:command_id>/',
views.edit_command, name='edit_command'),
]
```

In the code above, we add a new URL pattern using the `path()` function. This route will accept `command_id` as a parameter indicating which command to edit. When users access the corresponding URL, for example, `/edit_command/1/`, Django will call the `edit_command` function that we defined earlier.

Defining a name for this route is also very important. In this case, we use `name='edit_command'`. This naming makes it easier for us to refer to this route in other parts of the application, such as when creating links to the edit command page.

By setting up this URL routing, we ensure that users can easily edit the commands they have previously created. This process helps create a better user experience and makes our application more responsive to user actions.

12.6 Integration With Activity Log Model

In bot platform development, tracking user activity is a crucial aspect, especially in the context of bot management. By integrating an activity log feature, we can monitor important actions performed by users, such as creating a bot, editing a bot, adding commands, or deleting a bot. This is not only useful for audit purposes but also provides better visibility regarding interactions happening within the application.

In this section, we will discuss how to integrate the activity log model when creating bots, editing bots, creating commands, and editing commands. Each time a user performs these actions, we will store their activity log. After that, we will display the logs on the dashboard.

We need to modify each view where users create or edit a bot or command to record their activity into the ActivityLog model.

12.6.1 Integration for Create Bot

First, we will integrate the activity log when a user creates a bot. When the user submits the form to create a new bot, the system will automatically save this activity into the previously defined activity log model. Thus, every time a bot is created, this activity will be recorded.

For example, after the bot is successfully created, we can add a line of code to save this activity in the log. Here is an implementation example in the `create_bot` function within the `views.py` file:

```
# File: apps/bots/views.py

from apps.dashboard.models import ActivityLog

@login_required
def create_bot(request):
    if request.method == 'POST':
        form = BotForm(request.POST)
        if form.is_valid():
            bot = form.save(commit=False)
            bot.user = request.user
            bot.is_active = True  # Set the bot to
active by default

            # Specify webhook URL according to the
platform
            if bot.bot_type == 'telegram':
                bot.webhook_url =
f'{WEBHOOK_URL}/telegram/'
            elif bot.bot_type == 'whatsapp':
                bot.webhook_url =
f'{WEBHOOK_URL}/whatsapp/'

            bot.save()
            # Set webhook based on bot type
            if bot.bot_type == 'telegram':

set_telegram_webhook(bot.token_telegram,
bot.webhook_url)
                # Add success message for Telegram
bot
                messages.success(request, 'Telegram
bot created successfully!')
            elif bot.bot_type == 'whatsapp':
                set_whatsapp_webhook(bot.webhook_url)
                # Add success message for WhatsApp
bot
                messages.success(request, f'WhatsApp
bot created successfully. Please set the following
webhook URL in your Twilio dashboard:
{bot.webhook_url}')

            # Add activity log
            ActivityLog.objects.create(
                user=request.user,
                action=f"Created a new bot:
{bot.name}"
            )

            return redirect('manage_bots')
    else:
        form = BotForm()

    return render(request, 'bots/create_bot.html',
{'form': form})
```

In the code above, after the bot is successfully created and saved to the database, we add the activity log using the `ActivityLog` model. This action records information such as the user who created the bot, a description of the activity, and a reference to the newly created bot object. All this information will be stored in the log table, making it easy to access at any time in the future.

12.6.2 Integration for Edit Bot

In addition to logging the bot creation activity, we also need to log when a user edits a bot. Just like creating a bot, when the user updates the bot information, this activity also needs to be logged.

Here is an implementation example for integrating the activity log within the `edit_bot` function:

```
@login_required
def edit_bot(request, bot_id):
    bot = get_object_or_404(Bot, id=bot_id)  # Get 
the bot based on ID
    if request.method == 'POST':
        form = BotForm(request.POST, instance=bot)  # 
Load data into the form
        if form.is_valid():
            form.save()  # Save the edited data

            # Add activity log after the bot is 
successfully edited
            ActivityLog.objects.create(
                user=request.user,
                action=f"Bot {bot.name} has been 
updated",
                bot=bot
            )

            return redirect('manage_bots')  # 
Redirect to the bot management page
    else:
        form = BotForm(instance=bot)  # If not POST, 
create form with bot data

    context = {
        'form': form,
        'bot': bot,
    }
    return render(request, 'bots/edit_bot.html', 
context)  # Display the template with context
```

12.6.3 Integration for Delete Bot

Deleting a bot from the bot creation platform is also an important activity that needs to be logged. Just like logging the creation and editing of bots, deleting a bot allows users and administrators to monitor changes made in the system. By logging this activity, we can review the deletion history performed by specific users, which is very useful for audit purposes and troubleshooting.

In this section, we will integrate the activity log feature into the bot deletion function. Each time a user deletes a bot, we will log information such as who deleted the bot and which bot was deleted.

The existing `manage_bots` function is responsible for displaying the list of bots owned by the user and also for handling bot deletion requests via POST. Let's add the activity log into this function so that each bot deletion can be logged properly.

Here is the implementation integrated with the activity log:

```
# File: apps/bots/views.py

@login_required  # Ensure that only logged-in users
can access this view
def manage_bots(request):
    bots = Bot.objects.filter(user=request.user)  #
Get the list of bots owned by the logged-in user

    if request.method == 'POST':
        if 'delete' in request.POST:  # Check if
there is a request to delete a bot
```

```
            bot_id = request.POST.get('delete')  #
Get the ID of the bot to be deleted from the form
            bot = get_object_or_404(Bot, id=bot_id)
# Find the bot by ID, or show 404 if not found

            # Save the deletion activity into the log
before the bot is deleted
            ActivityLog.objects.create(
                user=request.user,
                action=f"Bot {bot.name} has been
deleted"
            )
            bot.delete()  # Delete the bot from the
database
            messages.success(request, 'Bot
successfully deleted.')  # Add message after the bot
is deleted
            return redirect('manage_bots')  # After
the bot is deleted, return to the bot management page

    return render(request, 'bots/manage_bots.html',
{'bots': bots})  # Render the page with the list of
bots owned by the user
```

In the code above, there are several key steps taken when a bot is deleted:

1. **Retrieving the deleted bot**: When the user clicks the 'delete' button on one of their owned bots, the system will receive the ID of that bot through POST. Then, with the `get_object_or_404` function, we obtain the `Bot` object corresponding to that ID.
2. **Logging the deletion activity**: Before the bot is deleted, we log the deletion action into the `ActivityLog` model. The stored information includes the user who performed the deletion, the action taken (in this case, bot deletion), and a reference to the deleted bot.

3. **Deleting the bot**: After logging the activity, the bot is then deleted from the database using the `delete()` method. After that, the user will be redirected back to the `manage_bots` page, which displays the list of bots they own.

In this way, each time a bot is deleted by a user, the activity will be recorded in the system, allowing administrators to monitor deletion activities that occur. The activity log provides additional visibility into the changes happening in the system, which is very important for maintaining the integrity and security of the application.

Integrating the activity log is an important step in ensuring that every significant action in the application is properly recorded.

12.6.4 Integration for Add Command

At this stage, we will integrate activity logging into the command addition feature. Every time a user adds a new command to the bot they created, that action will be recorded in the activity log, ensuring that every significant change in the system can be traced. By logging each command addition, administrators and users can better monitor how interactions are conducted with the existing bots.

In this process, we will use the `ActivityLog` model to record information related to who added the command, which command was added, and to which bot the command was added. This helps provide full visibility into all interactions that occur within the application.

The code below shows how the command addition feature has been integrated with activity logging:

```
# File: apps/bots/views.py

@login_required
def add_command(request, bot_id):
    bot = get_object_or_404(Bot, id=bot_id,
user=request.user)  # Retrieve the bot by ID and
ensure the user is the owner of the bot

    if request.method == 'POST':
        form = BotCommandForm(request.POST)  # Create
a form based on POST data
        if form.is_valid():
            command = form.save(commit=False)  # Save
the new command but not yet to the database
            command.bot = bot  # Assign the related
bot to the command
            command.save()  # Save the command to the
database

            # Log the command addition activity
            ActivityLog.objects.create(
                user=request.user,
                action=f"Added new command:
{command.command} for bot {bot.name}"
            )

            return redirect('manage_bots',
bot_id=bot.id)  # After successfully saving, redirect
to the edit bot page
    else:
        form = BotCommandForm()  # If GET, display an
empty form
```

```
    return render(request, 'bots/add_command.html',
{'form': form, 'bot': bot})  # Render template with
the form
```

In the code above, the activity logging integration is carried out through several key steps:

1. ***Getting the bot object:*** By using `get_object_or_404`, the system ensures that the bot for which the command is to be added actually exists and that the user performing the action is the owner of that bot. This is an important step in ensuring that no other users can add commands to a bot that does not belong to them.
2. ***Form processing:*** If the request method is POST, the form filled out by the user will be processed and validated. If the form validation is successful, the command object will be saved to the database using `form.save(commit=False)`, allowing us to add additional properties (such as the related bot) before the command is actually saved.
3. ***Logging activity:*** After the command is successfully saved, we log the command addition action into the `ActivityLog` model. This log records information such as which user added the command and which command was added to the bot. By logging this information, we ensure that every significant change to the bot can be easily tracked.
4. ***Redirecting to the bot edit page:*** After the command has been successfully added and the activity logged, the user will be redirected back to the edit_bot page for that bot,

allowing them to immediately see the new command they have added.

This command addition feature integrated with activity logging provides users with the ability to monitor every action that occurs within the application, whether for auditing or troubleshooting purposes. Users and administrators can now know exactly when and by whom new commands were added to the bot.

12.6.5 Integration for Edit Command

The process of editing a command within a bot is a very important feature, as previously added commands may need to be updated to adjust to changing user needs or bot functionality. To ensure that every change to these commands is well documented, we need to integrate activity logging into the edit command feature. This allows us to know when and by whom a command has been changed, providing better transparency within the system.

In this case, when a user edits a command, the application will log the changes in the `ActivityLog` model. This way, every editing action taken can be tracked and monitored. This process is very similar to the command addition feature, but here we focus on editing existing commands.

The following code explains how we implement the command editing feature integrated with activity logging:

```
# File: apps/bots/views.py
```

```
@login_required
def edit_command(request, command_id):
    command = get_object_or_404(BotCommand,
id=command_id)  # Retrieve the command by ID
    bot = command.bot  # Get the bot related to the
command

    if request.method == 'POST':
        form = BotCommandForm(request.POST,
instance=command)  # Load POST data into the form
        if form.is_valid():
            form.save()  # Save the command changes
to the database

            # Log the command editing activity
            ActivityLog.objects.create(
                user=request.user,
                action=f"Edited command:
{command.command} in bot {bot.name}"
            )

            return redirect('manage_bots',
bot_id=bot.id)  # After successfully saving, redirect
back to the edit bot page
    else:
        form = BotCommandForm(instance=command)  # If
not POST, display the form with the existing command
data

    return render(request, 'bots/edit_command.html',
{'form': form, 'bot': bot})  # Render template with
the form and related bot
```

In the code above, there are several steps to note in this integration process:

1. ***Retrieving the command and bot objects:*** The system retrieves the command to be edited by its ID using `get_object_or_404`. This way, we ensure that the command to be edited actually exists and is valid. Then, the bot associated with that command is also retrieved so

that we can display the bot's information to the user and log the editing activity for that specific bot.

2. ***Form processing:*** Similar to the command addition feature, when the user submits the form via the POST method, the system will validate the data entered. If the form is valid, the changes made to the command will be saved to the database.
3. ***Logging activity:*** After the command changes have been successfully saved, we immediately log that activity into the `ActivityLog` model. The information recorded includes who edited the command, which command was edited, and which bot the command belongs to. This logging is important to provide a clear audit trail of every change made in the application.
4. ***Redirecting after saving:*** After the command has been successfully changed, the user will be redirected back to the edit bot page, where they can see the list of commands that have just been modified.

With the integration of activity logging for the edit command feature, every change made by users is well documented. Other users and application administrators can monitor who has made changes, as well as which commands have been altered, ensuring better control and management of the bots and commands within the system.

12.6.6 Integration for Delete Command

```
@login_required
def manage_bots(request):
    bots = Bot.objects.filter(user=request.user)  #
Display only bots owned by the currently logged-in
user

    if 'delete_command' in request.POST:
        command_id =
request.POST.get('delete_command')  # Get the ID of
the command to be deleted
        command = get_object_or_404(BotCommand,
id=command_id)  # Get the command by ID

        # Log the deletion activity of the command
        ActivityLog.objects.create(
            user=request.user,
            action=f"Deleting command:
{command.command} from bot {command.bot.name}"
        )

        command.delete()  # Delete the command from
the database
        return redirect('manage_bots')  # Redirect
back to the manage bots page

    return render(request, 'bots/manage_bots.html',
{'bots': bots})
```

With the steps above, we have successfully integrated the activity log for every action of creating and editing bots, creating commands, and editing commands, as well as displaying them on the dashboard.

12.7 Complete Code For Bots

I will review the complete code in our bots application to avoid errors. Here is the complete code for everything:

12.7.1 Models.py

```
# File: apps/bots/models.py

from django.db import models
from django.contrib.auth.models import User

class Bot(models.Model):
    BOT_CHOICES = [
        ('telegram', 'Telegram'),
        ('whatsapp', 'WhatsApp'),
    ]
    user = models.ForeignKey(User,
on_delete=models.CASCADE)  # Bot will be linked to
the user
    bot_type = models.CharField(max_length=20,
choices=BOT_CHOICES)  # Type of bot: Telegram or
WhatsApp

    token_telegram = models.CharField(max_length=255,
unique=True, help_text="Enter a unique bot token.")
# Telegram token
    webhook_url = models.URLField(max_length=255,
blank=True, help_text="The webhook URL will be set
automatically.")  # Webhook URL

    is_active = models.BooleanField(default=True,
help_text="Whether the bot is active or not.")  #
Active status of the bot

    # Automatic messages
    start_message = models.TextField(blank=True,
default="Welcome to the bot!", help_text="Message
sent when the user types /start.")
    help_message = models.TextField(blank=True,
default="Here’s how to use the bot.",
help_text="Message sent when the user types /help.")

    messages = models.JSONField(default=list,
blank=True, help_text="List of messages received and
their responses in JSON format.")  # JSON format
```

```
    # Timestamp
    created_at =
models.DateTimeField(auto_now_add=True)  # Date the
bot was created
    updated_at = models.DateTimeField(auto_now=True)
# Date the bot was last updated

    name = models.CharField(max_length=255,
blank=True, default="My Bot", help_text="Name of the
bot.")  # Bot name
    description = models.TextField(blank=True,
default="This is my first bot",
help_text="Description of the bot.")  # Bot
description

    def __str__(self):
        return f'{self.bot_type.capitalize()} Bot -
{self.user.username}'

    class Meta:
        verbose_name = "Bot"
        verbose_name_plural = "Bots"

class BotCommand(models.Model):
    bot = models.ForeignKey(Bot,
related_name='commands', on_delete=models.CASCADE)
    command = models.CharField(max_length=255,
help_text="Enter commands like '/start', 'hello',
etc.")
    response = models.TextField(help_text="Enter the
response that will be sent when the command is
received.")

    def __str__(self):
        return f"Command: {self.command} -> Response:
{self.response}"

    class Meta:
        unique_together = ['bot', 'command']  #
Ensure each bot only has one command with the same
name.
        verbose_name = "Bot Command"
        verbose_name_plural = "Bot Commands"
```

12.7.2 Forms.py

```
# File: apps/bots/forms.py

from django import forms
from django.core.exceptions import ValidationError
import re  # For token pattern validation
from .models import Bot, BotCommand  # Importing Bot
and BotCommand models

class BotForm(forms.ModelForm):
    class Meta:
        model = Bot  # Link the form with the Bot
model
        fields = ['bot_type', 'token_telegram',
'name', 'description', 'is_active', 'start_message',
'help_message']  # Add start_message and help_message
        widgets = {
            'name': forms.TextInput(attrs={'class':
'form-control', 'placeholder': 'Enter bot name'}),
            'description':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Bot description'}),
            'start_message':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Message sent when
/start'}),
            'help_message':
forms.Textarea(attrs={'class': 'form-control',
'rows': 3, 'placeholder': 'Message sent when
/help'}),
        }
        help_texts = {
            'token_telegram': 'Enter the unique token
received from the Telegram platform.',
            'name': 'The bot name can be changed as
desired.',
            'description': 'A brief description of
the bot and its function.',
            'start_message': 'The message sent when
the user types /start.',
            'help_message': 'The message sent when
the user types /help.',
        }

    # Custom validation for Telegram token
    def clean_token_telegram(self):
        token_telegram =
self.cleaned_data.get('token_telegram')
        bot_type = self.cleaned_data.get('bot_type')

        # Validate only if bot_type is Telegram
        if bot_type == 'telegram':
            # Telegram token pattern: digits followed
by a colon, then a series of random characters
            if token_telegram and not re.match(r'^\
d+:[\w-]+$', token_telegram):
                raise ValidationError("Invalid
Telegram token. Ensure the token format is like
'123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11'.")

        return token_telegram

    # Bot name validation
    def clean_name(self):
        name = self.cleaned_data.get('name')
        if not name.strip():
            raise ValidationError("The bot name
cannot be empty.")
        return name

    # Bot description validation
    def clean_description(self):
        description =
self.cleaned_data.get('description')
        if len(description) > 500:
            raise ValidationError("The bot
description cannot be more than 500 characters.")
        return description

class BotCommandForm(forms.ModelForm):
    class Meta:
        model = BotCommand
        fields = ['command', 'response']  # Fields to
be displayed in the form
        widgets = {
            'command':
forms.TextInput(attrs={'placeholder': 'Example:
/start, /help'}),
            'response': forms.Textarea(attrs={'rows':
3, 'placeholder': 'Enter the response to be sent.'}),
        }
        help_texts = {
            'command': 'Enter commands like "/start",
"/help", or others.',
            'response': 'Enter the response that will
be sent when the command is received.',
        }
```

12.7.3 Views.py

```
# File: apps/bots/views.py

from django.shortcuts import render, redirect,
get_object_or_404
from django.contrib.auth.decorators import
login_required
from django.contrib import messages
from .forms import BotForm, BotCommandForm
from .models import Bot, BotCommand
from django.conf import settings
from apps.webhook.views import set_telegram_webhook
from apps.webhook.views import set_whatsapp_webhook
from apps.dashboard.models import ActivityLog

WEBHOOK_URL = settings.WEBHOOK_URL

@login_required
def create_bot(request):
    if request.method == 'POST':
        form = BotForm(request.POST)
        if form.is_valid():
            bot = form.save(commit=False)
            bot.user = request.user
            bot.is_active = True  # Set bot active by
default

            # Set webhook URL according to the
platform
            if bot.bot_type == 'telegram':
                bot.webhook_url =
f'{WEBHOOK_URL}/telegram/'
            elif bot.bot_type == 'whatsapp':
                bot.webhook_url =
f'{WEBHOOK_URL}/whatsapp/'

            bot.save()

            # Set webhook based on bot type
            if bot.bot_type == 'telegram':

set_telegram_webhook(bot.token_telegram,
bot.webhook_url)
                # Add success message for Telegram
bot
                messages.success(request, 'Telegram
bot successfully created!')
            elif bot.bot_type == 'whatsapp':
                set_whatsapp_webhook(bot.webhook_url)
                # Add success message for WhatsApp
bot
                messages.success(request, f'WhatsApp
bot successfully created. Please set the following
webhook URL in your Twilio dashboard:
{bot.webhook_url}')

            # Add activity log
            ActivityLog.objects.create(
                user=request.user,
                action=f"Created new bot: {bot.name}"
            )

            return redirect('manage_bots')
    else:
        form = BotForm()
    return render(request, 'bots/create_bot.html',
{'form': form})

@login_required  # Ensure only logged-in users can
access this view
def manage_bots(request):
    bots = Bot.objects.filter(user=request.user)  #
Get list of bots owned by the logged-in user

    if request.method == 'POST':
        if 'delete' in request.POST:  # Check if
there is a request to delete a bot
            bot_id = request.POST.get('delete')  #
Get the ID of the bot to delete from the form
            bot = get_object_or_404(Bot, id=bot_id,
user=request.user)  # Ensure the bot belongs to the
logged-in user

            # Log the deletion activity before
deleting the bot
            ActivityLog.objects.create(
                user=request.user,
                action=f"Bot {bot.name} has been
deleted"
            )
            bot.delete()  # Delete the bot from the
database
            messages.success(request, 'Bot
successfully deleted.')  # Add message after bot is
deleted
            return redirect('manage_bots')  # After
deleting the bot, return to the manage bots page

        elif 'delete_command' in request.POST:
            command_id =
request.POST.get('delete_command')  # Get the ID of
the command to delete
            command = get_object_or_404(BotCommand,
id=command_id)  # Get the command by ID

            # Ensure the command belongs to the
user's bot
            if command.bot.user == request.user:
                # Log the deletion of the command
                ActivityLog.objects.create(
                    user=request.user,
                    action=f"Deleting command:
{command.command} from bot {command.bot.name}"
                )
                command.delete()  # Delete the
command from the database
                messages.success(request, 'Command
successfully deleted.')  # Add message after command
is deleted
            else:
                messages.error(request, 'Access
denied. This command does not belong to you.')

            return redirect('manage_bots')  #
Redirect back to the manage bots page

    return render(request, 'bots/manage_bots.html',
{'bots': bots})  # Render the page with the list of
bots owned by the user

@login_required
def edit_bot(request, bot_id):
    bot = get_object_or_404(Bot, id=bot_id)  # Get
the bot by ID
    if request.method == 'POST':
        form = BotForm(request.POST, instance=bot)  #
Load data into the form
        if form.is_valid():
            form.save()  # Save the edited data

            # Add activity log after the bot has been
successfully edited
            ActivityLog.objects.create(
                user=request.user,
                action=f"Bot {bot.name} has been
updated",
                bot=bot
            )

            return redirect('manage_bots')  #
Redirect to the manage bots page
```

```
    else:
        form = BotForm(instance=bot)  # If not POST, 
create form with bot data

    context = {
        'form': form,
        'bot': bot,
    }
    return render(request, 'bots/edit_bot.html', 
context)  # Show template with context

@login_required
def add_command(request, bot_id):
    bot = get_object_or_404(Bot, id=bot_id, 
user=request.user)  # Get the bot by ID and ensure 
the user is the owner

    if request.method == 'POST':
        form = BotCommandForm(request.POST)  # Create 
form based on POST data
        if form.is_valid():
            command = form.save(commit=False)  # Save 
new command but not to database yet
            command.bot = bot  # Assign the related 
bot to the command
            command.save()  # Save the command to the 
database

            # Log the addition of the command
            ActivityLog.objects.create(
                user=request.user,
                action=f"Adding new command: 
{command.command} for bot {bot.name}"
            )

            return redirect('manage_bots')  # After 
successfully saving, redirect to the edit bot page
    else:
        form = BotCommandForm()  # If GET, display an 
empty form
```

```
    return render(request, 'bots/add_command.html',
{'form': form, 'bot': bot})  # Render template with
form

@login_required
def edit_command(request, command_id):
    command = get_object_or_404(BotCommand,
id=command_id)  # Get the command by ID
    bot = command.bot  # Get the bot related to the
command

    if request.method == 'POST':
        form = BotCommandForm(request.POST,
instance=command)  # Load POST data into the form
        if form.is_valid():
            form.save()  # Save changes to the
command in the database

            # Log the editing of the command
            ActivityLog.objects.create(
                user=request.user,
                action=f"Editing command:
{command.command} in bot {bot.name}"
            )

            return redirect('manage_bots')  # After
successfully saving, redirect back to the edit bot
page
    else:
        form = BotCommandForm(instance=command)  # If
not POST, display form with existing command data

    return render(request, 'bots/edit_command.html',
{'form': form, 'bot': bot})  # Render template with
form and related bot
```

12.7.4 Services/telegram.py

```
# File: apps/bots/services/telegram.py

from apps.bots.models import Bot, BotCommand
```

```
import requests

def handle_update(update):
    chat_id = update['message']['chat']['id']
    text = update['message']['text'].lower()  #
Convert to lowercase for consistent comparison

    bot =
Bot.objects.filter(bot_type='telegram').first()  #
Assume we take the first bot, can be adjusted

    if bot:
        # Check if the command matches what is in the
BotCommand model
        command = BotCommand.objects.filter(bot=bot,
command__iexact=text).first()

        if command:
            response_text = command.response
        else:
            # If the command is not found, use
default commands /start and /help
            if text == "/start":
                response_text = bot.start_message
            elif text == "/help":
                response_text = bot.help_message
            else:
                response_text = 'Unknown command.
Type /help to see the list of commands.'

        send_message(bot.token, "sendMessage", {
            'chat_id': chat_id,
            'text': response_text
        })

def send_message(token, method, data):
    telegram_api_url =
f'https://api.telegram.org/bot{token}/{method}'
    response = requests.post(telegram_api_url,
data=data)
    if response.status_code != 200:
        logger.error(f"Failed to send message:
{response.text}")  # Log error if failed
```

```
    return response
```

12.7.5 Services/whatsapp.py

```
# File: apps/bots/services/whatsapp.py

from django.http import HttpResponse  # For returning
HTTP response
from django.views.decorators.csrf import csrf_exempt
# To allow the view to be accessed without CSRF
validation
from twilio.twiml.messaging_response import
MessagingResponse  # To create a response to Twilio
API
from apps.bots.models import Bot, BotCommand  #
Import Bot and BotCommand models from the bots app
import logging  # For logging activities and errors

logger = logging.getLogger(__name__)  # Initialize
logger

@csrf_exempt  # Decorator to disable CSRF validation
on this view
def handle_update(update):
    # Get the sender of the message and the message
content
    user = update['From']
    message = update['Body'].strip().lower()  #
Normalize the message to lowercase

    # Fetch the WhatsApp bot based on the sender
number (you can adjust this logic)
    bot =
Bot.objects.filter(bot_type='whatsapp').first()

    if bot:
        # Check for commands based on the message
sent by the user
        command = BotCommand.objects.filter(bot=bot,
command__iexact=message).first()
```

```
        if command:
            response_text = command.response  # Get
the response associated with the command
        else:
            # Default message if no command matches
            response_text = 'Unknown command. Type
/help to see the list of commands.'

        # Send the response message to the user
        twilio_response = MessagingResponse()  #
Create a new MessagingResponse object
        twilio_response.message(response_text)  # Add
the response message to the Twilio response
        return HttpResponse(str(twilio_response),
content_type='text/xml')  # Return the Twilio
response

    # Log if no bot was found
    logger.warning('No WhatsApp bot found for
handling the message.')
    return HttpResponse(status=200)  # Respond with
200 OK if no bot is found
```

Chapter Conclusion

In this chapter, we discussed in-depth the command feature in the bot within the platform_bot project. Commands in a bot are special instructions given by users to trigger specific responses or actions from the bot. This feature is an essential element in bot interactions, enabling the bot to respond to various user inputs in a structured and efficient manner.

In this section, we have implemented two main features:

1. ***Adding Commands***: This feature allows users to add new commands to their bot. We created a model to store command data, a form for inputting new commands, and views and templates to handle the process of adding commands. With this feature, users can expand the functionality of their bots with new commands that suit their needs.
2. ***Editing Commands***: This feature allows users to modify existing commands. We have created views and templates for the command editing process, as well as set up URL routing to ensure that users can easily access the editing page. Editing commands provides additional flexibility in managing existing commands, allowing adjustments to bot responses according to changing needs.

Keys to Success in Adding and Editing Commands

Success in adding and editing commands in the bot heavily depends on several key factors:

1. ***Efficient Model Design***: A well-designed `BotCommand` model ensures that command data is stored correctly and structured. By using the `unique_together` attribute, we prevent command duplication for the same bot, maintaining data integrity.
2. ***User-Friendly Forms***: The `BotCommandForm` is designed to facilitate users in entering and editing commands. By utilizing widgets such as `TextInput` and `Textarea`, and adding clear help text, this form provides an intuitive user experience.
3. ***Proper View Implementation***: Views for adding and editing commands ensure that data from the form is processed correctly. By handling POST requests to save data and GET requests to display the form, these views facilitate user interaction with the system.
4. ***Clear HTML Templates***: The `add_command.html` and `edit_command.html` templates are designed to provide a clean and easily understood user interface. With the use of clear form elements and good layout arrangements, these templates enhance the user experience when adding or editing commands.
5. ***Consistent URL Routing***: Proper URL routing setup ensures that users can easily access the add and edit command features. Consistent and intuitive routing helps keep application navigation simple and accessible.

By correctly implementing these features, you ensure that the bot can function better and respond to user needs. Effective and easily manageable command features are key to enhancing the functionality and usability of the bot in your platform.

Chapter 13 - Refining the Home Page

In this chapter, we will focus on refining the home page of our `platform_bot` project using Bootstrap. Bootstrap is a highly popular CSS framework widely used in web development. This framework offers a variety of components and utilities that make it easier to create responsive and attractive web designs. The goal of this chapter is to leverage Bootstrap's features to enhance the look and user experience of `platform_bot`, making it more professional and user-friendly.

13.1 Introduction

13.1.1 Purpose and Benefits of Using Bootstrap

Bootstrap is a valuable tool in web development for several reasons. First, Bootstrap provides a variety of ready-to-use UI components, such as buttons, forms, tables, and navigation, which can help accelerate the development process. Second, with its grid system and utility classes, Bootstrap enables the creation of responsive designs that automatically adapt to various screen sizes, from desktops to mobile devices. This is important to ensure that `platform_bot` is accessible and usable comfortably across different devices. Third, Bootstrap offers visual consistency that can help maintain a uniform appearance for the user interface across the entire application.

13.1.2 Basic Introduction to Bootstrap

Bootstrap was developed by Twitter and released as an open-source project. This framework offers several basic components that can be used directly in our projects.

Here are some key features of Bootstrap:

1. ***Grid System***: The Bootstrap grid system allows us to create responsive layouts using predefined classes. This grid divides the page into columns that can be arranged for various screen sizes.
2. ***UI Components***: Bootstrap provides various components such as buttons, forms, tables, alerts, modals, and navigation, which can be used to speed up the process of creating user interfaces.
3. ***Utilities***: Bootstrap offers utility classes to quickly make various style adjustments, such as margins, padding, colors, and element visibility.
4. ***Customization***: Bootstrap also supports theme customization by providing SASS files that allow us to change colors, sizes, and styles according to project needs.

13.1.3 Installing and Configuring Bootstrap in a Django Project

To start using Bootstrap in our Django project, we first need to install and configure it. Here are the steps to do so:

Installing Bootstrap via CDN

The easiest way to integrate Bootstrap into our project is by using a Content Delivery Network (CDN). We just need to add links to the Bootstrap CSS and JavaScript files in our HTML template.

Add the following code in the `<head>` section of the `base.html` template:

```
<!-- Link to Bootstrap CSS -->
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.
5.2/css/bootstrap.min.css">

<!-- Link to Bootstrap JavaScript -->
<script src="https://code.jquery.com/jquery-
3.5.1.slim.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.
3/dist/umd/popper.min.js"></script>
<script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5
.2/js/bootstrap.min.js"></script>
```

Installing Bootstrap via NPM

If you prefer to manage dependencies using NPM, you can install Bootstrap with the following command. First, ensure you are in your Django project directory, then run:

```
$ npm install bootstrap
```

After that, add Bootstrap to your project's CSS and JavaScript files. For example, add it to `static/css/styles.css` for CSS and `static/js/scripts.js` for JavaScript:

```
/* Adding Bootstrap to the CSS file */
@import "~bootstrap/dist/css/bootstrap.min.css";

javascript
Copy code
// Adding Bootstrap to the JavaScript file
import 'bootstrap';
```

Configuring the HTML Template

Once Bootstrap is installed, we need to update our HTML template to use Bootstrap components and utilities. Let's look at some examples of using Bootstrap on existing pages:

Home Page (`home.html`)

Add Bootstrap classes to the HTML elements to create a responsive and attractive layout. For example, use the `container` class to wrap content and the `row` and `col` classes to arrange columns:

```
<div class="container">
  <div class="row">
    <div class="col-md-6">
      <h1>Welcome to the Bot Platform</h1>
      <p>Our Bot Platform offers various features to
manage your Telegram bots.</p>
    </div>
    <div class="col-md-6">
      <img src="#" class="img-fluid" alt="Welcome">
    </div>
  </div>
</div>
```

By using Bootstrap, we can easily improve and beautify the user interface of our `platform_bot`. Next, we will utilize Bootstrap to further refine other pages and ensure that each part of the application looks professional and functions well across various devices.

13.2 Improving The Home Page

Currently, the home page of our platform_bot consists of the files `base.html` and `home.html`. The `base.html` file serves as a basic template that includes common elements such as the navbar, footer, and links to the Bootstrap CSS and JavaScript files. This file establishes the basic structure of the page and ensures design consistency throughout our application.

The `home.html` file extends `base.html` and provides content specific to the main page, such as a welcome title and a description of the application. However, the current appearance is still simple and does not fully utilize Bootstrap features to create a responsive and attractive design. In this section, we will leverage various Bootstrap features to enhance the appearance of our home page.

13.2.1 Updating `base.html`

We will update `base.html` by adding more Bootstrap features and icons from Bootstrap Icons to improve the appearance and functionality of the page. We will enhance the `base.html` file to more effectively utilize Bootstrap components and add some additional visual elements.

Here, we will change the navbar color to the primary color, add backgrounds to several sections, and adjust the footer.

Below are the updates for the `base.html` file:

```html
<!-- File: apps/templates/home/base.html -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>Platform Bot</title>
    <!-- Link to Bootstrap CSS -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhj
Y6hW+ALEwIH" crossorigin="anonymous">
    <!-- Link to Bootstrap Icons -->
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.10.5/font/bootstrap-icons.css"
rel="stylesheet">
</head>
<body class="d-flex flex-column min-vh-100">
    <!-- Header -->
    <nav class="navbar navbar-expand-lg navbar-dark
bg-primary shadow-sm">
        <div class="container">
            <a class="navbar-brand d-flex align-
items-center" href="{% url 'home' %}">
                <i class="bi bi-lightning-charge-fill
me-2"></i>Platform Bot
            </a>
            <button class="navbar-toggler"
type="button" data-bs-toggle="collapse" data-bs-
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 ms-auto mb-2
mb-lg-0">
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'home' %}"><i class="bi bi-house-fill me-
1"></i>Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'login' %}"><i class="bi bi-box-arrow-in-right
me-1"></i>Login</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'register' %}"><i class="bi bi-person-plus-fill
me-1"></i>Register</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'dashboard' %}"><i class="bi bi-speedometer2 me-
1"></i>Dashboard</a>
                    </li>
                </ul>
                <form class="d-flex ms-lg-3"
role="search">
                    <input class="form-control me-2"
type="search" placeholder="Search" aria-
label="Search">
                    <button class="btn btn-outline-
light" type="submit">
                        <i class="bi bi-search"></i>
                    </button>
                </form>
            </div>
        </div>
    </nav>

    <!-- Main Content -->
    <div class="container mt-4">
        {% block content %}
        {% endblock %}
    </div>

    <!-- Footer -->
    <footer class="footer mt-auto py-4 bg-dark text-
white-50">
        <div class="container text-center">
            <span>© 2024 Platform Bot</span>
            <div class="mt-3">
                <a href="#" class="btn btn-outline-
light btn-sm me-2">
                    <i class="bi bi-facebook"></i>
                </a>
                <a href="#" class="btn btn-outline-
light btn-sm me-2">
                    <i class="bi bi-twitter"></i>
                </a>
                <a href="#" class="btn btn-outline-
light btn-sm">
                    <i class="bi bi-instagram"></i>
                </a>
            </div>
        </div>
    </footer>

    <!-- Bootstrap JS and dependencies -->
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11
.8/dist/umd/popper.min.js" integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc
2pM8ODewa9r" crossorigin="anonymous"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/js/bootstrap.min.js" integrity="sha384-
0pUGZvbkm6XF6gxjEnlmuGrJXVbNuzT9qBBavbLwCsOGabYfZo0T0
to5eqruptLy" crossorigin="anonymous"></script>
</body>
</html>
```

Explanation of Changes:

1. ***Responsive Navbar***: Added a toggler button to the navbar for a responsive view on mobile devices. This allows the navbar menu to switch between vertical and horizontal views on various screen sizes.
2. ***Search Form***: Added a search form in the navbar with a search icon from Bootstrap Icons. This form provides users with the option to search for content on the site.
3. ***Social Media Icons***: Added social media icons in the footer using buttons with icons from Bootstrap Icons. This enhances the site's social connectivity.
4. ***Container Usage***: Used the container class to align elements in the navbar and footer for a neater and more consistent appearance across different devices.
5. ***Responsiveness***: Ensured that all elements, such as the navbar and footer, remain responsive and display well on various screen sizes.

With these updates, `base.html` becomes more functional and aesthetic, leveraging Bootstrap features for responsive design and icons that enrich the appearance. Next, ensure to test the view on various devices and adjust any additional CSS if needed.

13.2.2 Adding Bootstrap Components

To further enhance the appearance of the home page, we will utilize various Bootstrap components such as the Navbar, Jumbotron, and Card.

The Navbar has been added in `base.html`, providing clear navigation for users. On the home page, we are using a

Jumbotron to display a welcome message with an eye-catching design. Additionally, we can add Card components to present additional information in a structured style.

Here is an example of using the Card component:

```
<!-- apps/templates/home/home.html -->

{% extends 'home/base.html' %}

{% block content %}
<div class="jumbotron bg-primary text-white rounded
p-4 mb-4 text-center">
    <h1 class="display-4">Welcome to the Bot
Platform!</h1>
    <p class="lead">Discover the best automation
solutions for your bot needs. Our platform offers
various advanced features to simplify your work.</p>
    <hr class="my-4">
    <p>Start by registering or logging in for full
access to our great features.</p>
    <a class="btn btn-light btn-lg" href="{% url
'register' %}" role="button">Get Started Now</a>
    <p class="mt-4">Already have an account? <a
class="text-muted" href="{% url 'login' %}">Log in
here</a></p> <!-- Link to Login page -->
</div>

<!-- Card Section -->
<div class="container mb-5">
    <div class="row">
        <div class="col-md-4 mb-4">
            <div class="card shadow-sm">
                <div class="card-body text-center">
                    <i class="bi bi-robot fs-2 mb-
3"></i>
                    <h5 class="card-title">Bot
Automation</h5>
```

```
                    <p class="card-text">Leverage our
automation capabilities to optimize routine tasks and
workflows with easily customizable bots.</p>
                    <a href="#" class="btn btn-
primary">Learn More</a>
                </div>
            </div>
        </div>
        <div class="col-md-4 mb-4">
            <div class="card shadow-sm">
                <div class="card-body text-center">
                    <i class="bi bi-gear fs-2 mb-
3"></i>
                    <h5 class="card-title">Flexible
Settings</h5>
                    <p class="card-text">Customize
every aspect of your bot with flexible settings and
an intuitive interface. Complete control is in your
hands.</p>
                    <a href="#" class="btn btn-
primary">Learn More</a>
                </div>
            </div>
        </div>
        <div class="col-md-4 mb-4">
            <div class="card shadow-sm">
                <div class="card-body text-center">
                    <i class="bi bi-bar-chart-line
fs-2 mb-3"></i>
                    <h5 class="card-title">In-Depth
Analytics</h5>
                    <p class="card-text">Gain deep
insights with our analytics features. Monitor your
bot's performance and make better decisions based on
data.</p>
                    <a href="#" class="btn btn-
primary">Learn More</a>
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

13.2.3 Refining Responsive Appearance

By using Bootstrap classes, our home page is already responsive, but we can make some additional adjustments to ensure an optimal appearance on various devices. Make sure that elements such as images, cards, and buttons work well on both small and large screens.

For example, in the card section we have added, we ensure that the size of the images and text can adjust according to the screen size. For images, we use the `img-fluid` class to ensure the images are responsive and do not overflow the container.

Explanation of Changes:

1. ***Navbar***: Changed the navbar color to `bg-primary` with white text for better contrast. Added a search form with a light-colored button to make it more prominent.
2. ***Footer***: Changed the background color of the footer to `bg-secondary` with white text and added social media icons with light colors to match the color scheme.
3. ***Home Page***:
   - Used a `bg-info` background with white text on the Jumbotron to highlight the welcome message with an appealing design.
   - Added a card section to showcase key features with images, titles, and brief descriptions. This gives a more dynamic and informative appearance.

With these updates, the appearance of `base.html` and `home.html` becomes more colorful, informative, and visually appealing while still maintaining a responsive and functional design.

13.2.4 Adding Content to the Home Page

We will add more content to the `home.html` page by changing the background of the jumbotron to `bg-primary`. We will add additional sections such as testimonials, latest news, and a call-to-action. This will make the homepage more informative and engaging.

Here’s the update for `home.html`, which includes additional sections for testimonials, latest news, and call-to-action. We will also change the background of the jumbotron to `bg-primary`.

```
<!-- File: apps/templates/home/home.html -->

{% extends 'home/base.html' %}

{% block content %}
<!-- Hero Section -->
<div class="jumbotron bg-primary text-white rounded
p-4 mb-5 text-center">
    <h1 class="display-4"><i class="bi bi-lightbulb-
fill me-2"></i>Welcome to the Bot Platform!</h1>
    <p class="lead">Discover the best automation
solutions for your bot needs. Our platform offers a
variety of advanced features to simplify your
work.</p>
    <hr class="my-4">
    <p>Start by signing up or logging in for full
access to our amazing features.</p>
```

```
    <a class="btn btn-light btn-lg mb-3" href="{% url
'register' %}" role="button">Get Started Now</a>
    <p class="mt-3">Already have an account? <a
class="text-light" href="{% url 'login' %}">Log in
here</a></p>
</div>

<!-- Feature Section -->
<div class="container mb-5">
    <h2 class="text-center mb-4">Our Featured
Features</h2>
    <div class="row">
        <div class="col-md-4 mb-4">
            <div class="card border-0 shadow-sm bg-
light h-100 text-center">
                <div class="card-body">
                    <i class="bi bi-robot fs-1 text-
primary mb-3"></i>
                    <h5 class="card-title">Bot
Automation</h5>
                    <p class="card-text">Leverage our
automation capabilities to optimize routine tasks and
workflows with easily customizable bots.</p>
                    <a href="#" class="btn btn-
primary">Learn More</a>
                </div>
            </div>
        </div>
        <div class="col-md-4 mb-4">
            <div class="card border-0 shadow-sm bg-
light h-100 text-center">
                <div class="card-body">
                    <i class="bi bi-gear fs-1 text-
primary mb-3"></i>
                    <h5 class="card-title">Flexible
Settings</h5>
                    <p class="card-text">Customize
every aspect of your bot with flexible settings and
an intuitive interface. Full control is in your
hands.</p>
                    <a href="#" class="btn btn-
primary">Learn More</a>
                </div>
            </div>
        </div>
        <div class="col-md-4 mb-4">
            <div class="card border-0 shadow-sm bg-
light h-100 text-center">
                <div class="card-body">
                    <i class="bi bi-bar-chart-line
fs-1 text-primary mb-3"></i>
                    <h5 class="card-title">In-Depth
Analytics</h5>
                    <p class="card-text">Gain deep
insights with our analytics features. Monitor your
bot's performance and make better decisions based on
data.</p>
                    <a href="#" class="btn btn-
primary">Learn More</a>
                </div>
            </div>
        </div>
    </div>
</div>

<!-- Testimonial Section -->
<div class="container mb-5">
    <h2 class="text-center mb-4">What Our Users
Say</h2>
    <div class="row">
        <div class="col-md-4 mb-4">
            <div class="card border-light shadow-sm
bg-white h-100">
                <div class="card-body">
                    <i class="bi bi-people-fill fs-2
text-primary mb-3"></i>
                    <p class="card-text">"The Bot
Platform has transformed the way we work. Its
advanced features and user-friendly interface have
made our jobs much more efficient."</p>
                    <footer class="blockquote-
footer">Jane Doe, <cite title="Source Title">CEO
ExampleCorp</cite></footer>
                </div>
            </div>
        </div>
        <div class="col-md-4 mb-4">
            <div class="card border-light shadow-sm
bg-white h-100">
                <div class="card-body">
                    <i class="bi bi-chat-left-quote
fs-2 text-primary mb-3"></i>
                    <p class="card-text">"I am very
impressed with the customer support and functionality
offered. This platform is extremely helpful in
automating daily tasks."</p>
                    <footer class="blockquote-
footer">John Smith, <cite title="Source
Title">Freelancer</cite></footer>
                </div>
            </div>
        </div>
        <div class="col-md-4 mb-4">
            <div class="card border-light shadow-sm
bg-white h-100">
                <div class="card-body">
                    <i class="bi bi-star-fill fs-2
text-primary mb-3"></i>
                    <p class="card-text">"It's very
easy to integrate with existing systems. This
platform truly meets our needs."</p>
                    <footer class="blockquote-
footer">Alice Johnson, <cite title="Source
Title">Product Manager</cite></footer>
                </div>
            </div>
        </div>
    </div>
</div>

<!-- Latest News Section -->
<div class="container mb-5">
    <h2 class="text-center mb-4">Latest News</h2>
    <div class="row text-center">
        <div class="col-md-6 mb-4">
            <div class="card border-0 shadow-sm bg-
info text-white h-100">
                <div class="card-body">
```

```
                    <i class="bi bi-info-circle fs-2
mb-3"></i>
                    <h5 class="card-title">Latest
Update: Feature X Released</h5>
                    <p class="card-text">We have just
launched Feature X that will simplify Process Y. Read
more about what's new in this version.</p>
                    <a href="#" class="btn btn-
light">Read More</a>
                </div>
            </div>
        </div>
        <div class="col-md-6 mb-4">
            <div class="card border-0 shadow-sm bg-
warning text-dark h-100">
                <div class="card-body">
                    <i class="bi bi-file-earmark-text
fs-2 mb-3"></i>
                    <h5 class="card-title">Complete
Guide: How to Use the Bot Platform</h5>
                    <p class="card-text">Follow our
guide to make the most of all the features the Bot
Platform has to offer. Get tips and tricks from our
experts.</p>
                    <a href="#" class="btn btn-
dark">Learn More</a>
                </div>
            </div>
        </div>
    </div>
</div>

<!-- Call to Action Section -->
<div class="bg-success text-white p-5 text-center">
    <h2>Ready to Get Started?</h2>
    <p class="lead">Sign up now for full access to
all our amazing features and start boosting your
productivity today!</p>
    <a class="btn btn-light btn-lg" href="{% url
'register' %}" role="button">Sign Up Now</a>
</div>
{% endblock %}
```

Adjustments made:

1. ***Hero Section:*** Added more color to the background and text. The login link text is now light-colored.
2. ***Feature Section:*** Used larger icons and a lighter background color to make the features stand out more.
3. ***Testimonial Section:*** Added icons to each testimonial to enrich the visualization and maintain a modern look.
4. ***News Section:*** Used colored cards to showcase the latest news, adding contrast between news items to draw attention.
5. ***Call to Action Section:*** Used a different background color (green) to create a sense of urgency and attract users.

With these changes, the homepage will have more valuable content and a more appealing visual layout, enhancing the user experience to be more informative and engaging.

13.2.5 Testing the Home Page Display

After making updates, the final step is to test the home page display on various devices to ensure everything works as expected. Check the appearance across different screen sizes, either using the developer tools in your browser or with physical devices.

To run the Django server and view the results, use the following command:

```
$ python manage.py runserver
```

Open your browser and navigate to http://127.0.0.1:8000/ to view the home page. Ensure that all elements, including the navbar, jumbotron, and card, display correctly and responsively. Make additional adjustments if necessary to ensure an optimal user experience across all devices.

With these steps, the home page of the `platform_bot` has been updated with a more professional and responsive appearance, thanks to the implementation of Bootstrap features. Next, we will apply similar techniques to other pages to ensure design consistency throughout the application.

13.3 Refining The Register Page

Currently, your register page uses a basic HTML structure with several form elements. However, the appearance and responsiveness of this page can be improved by implementing the Bootstrap grid system and adding Bootstrap components to enhance its appearance and functionality.

13.3.1 Integrating the Bootstrap Grid System

The Bootstrap Grid System makes it easy to create a flexible, responsive layout. On the register page, we will use the grid to ensure that the form can adapt to various screen sizes.

To integrate the grid system, pay attention to the existing structure within the `<div class="container">`. We will optimize the use of columns by adding Bootstrap grid classes.

13.3.2 Adding Bootstrap Components

We will update the register page to use Bootstrap components such as card, form-group, and input-group to make the form appear more modern and responsive.

To enhance the appearance of this registration page to look more professional, we will add a background to the form, more icons, and utilize components and colors available in Bootstrap 5 so that the page looks more modern. Remove specific styling like border-radius that uses inline styles, and replace it with Bootstrap classes.

I will make the following adjustments:

1. Use the `bg-light` background color on the card with an additional border to make the form stand out more.
2. Add more Bootstrap icons to give an interactive feel to the form labels.
3. Change some text to `text-secondary` color for a softer and more professional look.
4. Utilize utility classes like `rounded`, `shadow`, and `p-4` for a neater and more structured layout.

Here are the results of the code improvements:

```
<!-- File: apps/templates/account/register.html -->
{% extends 'home/base.html' %}

{% block title %}Register - Bot Platform{% endblock
%}
```

```
{% block content %}
<div class="container mt-5">
    <!-- Header with Background and Text -->
    <h2 class="text-center bg-primary text-white py-2
rounded">New User Registration</h2>

    <!-- Form Card with Background and Shadow -->
    <div class="card mt-4 bg-light shadow-lg">
        <div class="card-body p-4">
            {% if form.errors %}
                <div class="alert alert-danger">
                    {% for field, errors in
form.errors.items %}
                        <strong>{{ field }}</strong>
                        {% for error in errors %}
                            <div>{{ error }}</div>
                        {% endfor %}
                    {% endfor %}
                </div>
            {% endif %}

            <!-- Form -->
            <form method="POST"
enctype="multipart/form-data">
                {% csrf_token %}

                <!-- First Name and Last Name -->
                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_first_name">
                            <i class="bi bi-person-
fill"></i> {{ form.first_name.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.first_name }}
                        </div>
                        {% if form.first_name.errors
%}
                            <div class="text-
danger">{{ form.first_name.errors }}</div>
                        {% endif %}
                    </div>
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_last_name">
                            <i class="bi bi-person-
fill"></i> {{ form.last_name.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.last_name }}
                        </div>
                        {% if form.last_name.errors
%}
                            <div class="text-
danger">{{ form.last_name.errors }}</div>
                        {% endif %}
                    </div>
                </div>

                <!-- Username and Email -->
                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_username">
                            <i class="bi bi-person-
circle"></i> {{ form.username.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.username }}
                        </div>
                        {% if form.username.errors %}
                            <div class="text-
danger">{{ form.username.errors }}</div>
                        {% endif %}
                    </div>
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_email">
                            <i class="bi bi-envelope-
fill"></i> {{ form.email.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.email }}
                        </div>
                        {% if form.email.errors %}
                            <div class="text-
danger">{{ form.email.errors }}</div>
                        {% endif %}
                    </div>
                </div>

                <!-- Password -->
                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_password1">
                            <i class="bi bi-lock-
fill"></i> {{ form.password1.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.password1 }}
                        </div>
                        {% if form.password1.errors
%}
                            <div class="text-
danger">{{ form.password1.errors }}</div>
                        {% endif %}
                    </div>
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_password2">
                            <i class="bi bi-lock-
fill"></i> {{ form.password2.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.password2 }}
                        </div>
                        {% if form.password2.errors
%}
                            <div class="text-
danger">{{ form.password2.errors }}</div>
```

```
                        {% endif %}
                    </div>
                </div>

                <!-- Company and Address -->
                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_company">
                            <i class="bi bi-
building"></i> {{ form.company.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.company }}
                        </div>
                        {% if form.company.errors %}
                            <div class="text-
danger">{{ form.company.errors }}</div>
                        {% endif %}
                    </div>
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_address">
                            <i class="bi bi-geo-alt-
fill"></i> {{ form.address.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.address }}
                        </div>
                        {% if form.address.errors %}
                            <div class="text-
danger">{{ form.address.errors }}</div>
                        {% endif %}
                    </div>
                </div>

                <!-- City and Country -->
                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_city">
```

```
                            <i class="bi bi-
building"></i> {{ form.city.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.city }}
                        </div>
                        {% if form.city.errors %}
                            <div class="text-
danger">{{ form.city.errors }}</div>
                        {% endif %}
                    </div>
                    <div class="col-md-6 mb-3">
                        <label class="fw-bold"
for="id_country">
                            <i class="bi bi-
globe"></i> {{ form.country.label }}
                        </label>
                        <div class="form-control
border border-primary">
                            {{ form.country }}
                        </div>
                        {% if form.country.errors %}
                            <div class="text-
danger">{{ form.country.errors }}</div>
                        {% endif %}
                    </div>
                </div>

                <!-- Submit Button -->
                <div class="text-center">
                    <button type="submit" class="btn
btn-primary my-3">
                        <i class="bi bi-person-plus-
fill"></i> Register
                    </button>
                    <br /><br />
                    <p>
                        Already have an account? <a
href="{% url 'login' %}" class="text-
primary">Login</a>
                    </p>
                </div>
```

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

Changes:

1. ***Background Card:*** I added `bg-light` and `shadow-lg` to the card to make it more attractive and professional.
2. ***Form Structure:*** Used more icons on input labels and maintained color consistency with `border border-primary` to highlight each form field.
3. ***Error Handling:*** If there are errors, the form will remain focused with red text color (`text-danger`) displayed under the field.
4. ***Button Style:*** The register button was updated with Bootstrap classes to maintain a consistent appearance.

Next, you can open the register page using the same Django run server command and observe the improvements made to the form's appearance. Ensure everything appears correctly across various screen sizes and make any necessary adjustments.

13.3.3 Testing the Register Page Appearance

To test the appearance of the register page, run the local Django server and open the register page in your browser:

```
$ python manage.py runserver
```

Open the URL http://localhost:8000/register/ in your browser and check if the register page displays as expected. Inspect the form elements to ensure all Bootstrap components are working well and the page layout is responsive across different screen sizes.

With this update, your register page not only looks more professional with Bootstrap integration but also provides a better and more responsive user experience.

13.4 Enhancing The Login Page

At this stage, we will enhance the `login.html` page to make it more attractive and responsive, providing a better user experience when logging into the bot platform.

13.4.1 Understanding the Current Structure of the Login Page

Before making modifications, we need to understand the existing structure of `login.html`. Currently, this page has a login form displayed in a card format, with elements like username and password.

The file being used now is `apps/templates/account/register.html`. In this file, the form structure is already quite good, but we will make some adjustments to make it look more appealing, using a fresher background color and a neater form layout.

13.4.2 Integrating the Bootstrap Grid System

One of the initial steps in enhancing the registration page's appearance is to utilize the Bootstrap Grid System. This grid system allows us to create a responsive layout according to the user's screen size.

Currently, the login form is already using `col-md-6` to set the column width, but we will add more control over the appearance on smaller devices. We will also use the `offset-md-3` class to ensure the form is centered on larger screens.

To create a more visually appealing login page by adding a background color and enhancing the form with icons, we can utilize Bootstrap elements to enrich the appearance while adjusting the colors and adding relevant icons. Below is an explanation and modification of the existing code.

Enhancement Steps:

1. ***Adding Background Color:*** We will use Bootstrap classes to add a background to the page. In the `<div class="content">`, add a background class with a soft color to be easy on the user's eyes.
2. ***Adding Icons to the Form:*** We will use icons from Bootstrap to add visuals to the input fields, making it easier for users to identify their functions.

Updated Login Page Code:

```
<!-- File: apps/templates/account/login.html -->

{% extends 'home/base.html' %}

{% block title %}Login - Platform Bot{% endblock %}

{% block content %}
<div class="content" style="background-color:
#f4f6f9; min-height: 100vh;">
    <div class="container">
        <!-- Display Django messages -->
        {% if messages %}
        <div class="row">
            <div class="col-md-6 offset-md-3">
                {% for message in messages %}
                <div class="alert alert-
{{ message.tags }}">
                    <button type="button"
class="close" data-dismiss="alert" aria-
label="Close">
                        <span aria-
hidden="true">&times;</span>
                    </button>
                    {{ message }}
                </div>
                {% endfor %}
            </div>
        </div>
        {% endif %}

        <div class="row pt-5">
            <div class="col-md-6 mt-5 offset-md-3 pt-
5 mt-5">
                <div class="card shadow-lg"
style="border-radius: 15px;">
                    <div class="card-header text-
center py-4 bg-primary text-white" style="border-
radius: 15px 15px 0 0;">
                        <h4 class="title"><i
class="bi bi-person-circle"></i> Login to Platform
Bot</h4>
                    </div>
                    <div class="card-body px-5 py-4">
                        <form method="post"
id="login-form">
                            {% csrf_token %}
                            <div class="form-group
mb-3">
                                <label class="fw-
bold" for="id_username">
                                    <i class="bi bi-
person-fill"></i> Username
                                </label>
                                <div class="input-
group">
                                    <div
class="input-group-prepend">
                                        <span
class="input-group-text bg-primary text-white">
                                            <i
class="bi bi-person-fill"></i>
                                        </span>
                                    </div>
                                    {{ form.username|
add_class:"form-control border border-primary" }}
                                </div>
                            </div>

                            <div class="form-group
mb-3">
                                <label class="fw-
bold" for="id_password">
                                    <i class="bi bi-
lock-fill"></i> Password
                                </label>
                                <div class="input-
group">
                                    <div
class="input-group-prepend">
                                        <span
class="input-group-text bg-primary text-white">
                                            <i
class="bi bi-lock-fill"></i>
                                        </span>
                                    </div>
                                    {{ form.password|
add_class:"form-control border border-primary" }}
                                </div>
                            </div>

                            <div class="text-center
mt-4">
                                <button type="submit"
class="btn btn-primary btn-lg">
                                    <i class="bi bi-
box-arrow-in-right"></i> Login
                                </button>
                            </div>
                        </form>
                    </div>
                    <div class="card-footer text-
center">
                        <p>Don't have an account?
                            <a href="{% url
'register' %}" class="text-primary fw-bold">
                                <i class="bi bi-
pencil-square"></i> Register
                            </a>
                        </p>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Key Changes:

1. ***Bootstrap Icons:*** Bootstrap icons were added to the header, input fields (username and password), and the registration link to enhance the appearance.
2. ***Colors and Bootstrap Components:*** Using `bg-primary` for the header and icons, and the input

elements with `form-control border border-primary` to make the form more colorful and neat.

3. ***Button Placement:*** The Login button uses an icon and a larger size for a more professional look.

With these changes, the login page will be visually more appealing, functional, and responsive.

Chapter Conclusion

In this chapter, we focused on refining the home page of the Django project by leveraging Bootstrap to enhance the appearance and responsiveness of the page.

Here is a summary of each subsection:

1. ***Introduction***
   - ***Purpose and Benefits of Using Bootstrap***: Bootstrap is a CSS framework that greatly aids in accelerating the design and development process of web interfaces by providing ready-to-use components and a responsive grid system.
   - ***Basic Introduction to Bootstrap***: Bootstrap offers various components and CSS utilities that allow for the quick creation of modern and responsive web designs.
   - ***Installation and Configuration of Bootstrap in Django Projects***: Installation can be done through CDN or by downloading files from the Bootstrap website. Configuration in Django involves setting up static files to ensure Bootstrap is correctly integrated into the project.
2. ***Refining the Home Page***
   - ***Improving base.html***: Changes were made to the base.html template to include Bootstrap and additional components, enhancing the page structure by adding the necessary stylesheets and scripts for better appearance.

- ***Adding Bootstrap Components***: Components such as cards and jumbotrons were added to enhance the look of the home page, providing more structured information and appealing visuals.
- ***Enhancing Responsive Appearance***: Adjustments were made to ensure the home page displays well on various devices, including the use of responsive Bootstrap classes to effectively arrange elements.
- ***Adding Content to the Home Page***: Additional content such as status messages, activity logs, and placeholder charts were integrated to provide richer information to users.
- ***Testing the Home Page Appearance***: After changes were applied, the home page appearance was tested to ensure all elements worked well and displayed according to the desired design.

3. ***Refining the Register Page***

- ***Integrating the Bootstrap Grid System***: The Bootstrap grid system was utilized to neatly arrange elements on the register page, creating a more structured layout.
- ***Adding Bootstrap Components***: Bootstrap components such as form controls and buttons were added to enhance the appearance and functionality of the register page.
- ***Testing the Register Page Appearance***: The appearance of the register page was tested to

ensure responsiveness and accuracy according to the design.

4. ***Refining the Login Page***
    - ***Understanding the Current Login Page Structure***: Assessing the existing login page structure to identify areas that need improvement.
    - ***Integrating the Bootstrap Grid System***: Implementing the Bootstrap grid system on the login page to ensure a neat and responsive appearance.

By implementing the steps outlined in this chapter, the main pages of the Django application can be significantly enhanced, offering a better user experience and a more modern design. The use of Bootstrap allows for the creation of a consistent and appealing interface with high efficiency.

Chapter 14 - Dashboard Page Enhancement

In this chapter, we will focus on enhancing the dashboard page of our bot platform project using Bootstrap. We have already discussed the explanations and advantages of Bootstrap previously.

Next, we will begin by refining the dashboard page. Before we improve it, we will separate the navbar, sidebar, etc., into separate files.

To separate components like the navbar and sidebar for better organization, we can use the concept of template inheritance in Django or partial views if using another framework. This step will help maintain cleanliness and order in the file structure and facilitate maintenance.

In building a web application, it is important to keep the code clean and modular, especially when working with frequently reused elements, such as the navbar and sidebar. In this chapter, we will discuss how to separate these components into a tidy directory structure using Django, a Python-based web framework.

14.1 Separating The Template Directory Structure

Before delving into component separation, we need to understand the directory structure we will use. A good structure facilitates file management and allows for the addition of new components without cluttering the main code. Below is the template directory structure we will create for the dashboard page:

```
apps/
    templates/
        dashboard/
            base.html
            navbar.html
            sidebar.html
            dashboard.html
```

Let's discuss each one:

- ***apps/***: This folder is the directory for your Django application project.
- ***templates/***: This directory is where all HTML template files will be stored. It is the default folder that Django uses to look for templates if you set it in the project settings.
- ***dashboard/***: A sub-directory within templates/ specifically dedicated to storing templates related to the dashboard page. By separating it into its own folder, you ensure that dashboard-specific templates do not mix with other templates you might create in other applications.
- ***base.html***: The main template that will serve as the basic framework for the entire dashboard page. Within it, you

will import components like the navbar and sidebar as well as the main content area.

- ***navbar.html***: A template dedicated to the navbar (top navigation menu). This component is usually uniform across the dashboard pages and can be called through base.html.
- ***sidebar.html***: A template for the sidebar (side navigation menu). Like the navbar, the sidebar is typically used consistently across many dashboard pages, making it very suitable for separation.
- ***dashboard.html***: The template for the main dashboard page that will inherit the framework from base.html.

14.1.1 Separating Base

The base.html file is the foundation of the entire dashboard page. Here, we will define the basic framework that includes core elements such as the navbar, sidebar, and a space for the main content. As a first step, let's create the base.html file in the dashboard/ directory:

```
<!-- File: apps/templates/dashboard/base.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>{% block title %}Dashboard{% endblock
%}</title>
                <!-- Adding Bootstrap and Bootstrap
Icons from CDN -->
                <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/di
st/css/bootstrap.min.css" rel="stylesheet">
```

```
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.10.5/font/bootstrap-icons.css"
rel="stylesheet">
  </head>
  <body>
    <!-- Include Navbar -->
    {% include 'dashboard/navbar.html' %}

    <!-- Include Sidebar -->
    {% include 'dashboard/sidebar.html' %}

    <!-- Main Content -->
    <main class="mt-5 pt-3">
      {% block content %}
      <!-- Page content will go here -->
      {% endblock %}
    </main>

                <!-- Including Bootstrap JavaScript
-->
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11
.8/dist/umd/popper.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dis
t/js/bootstrap.min.js"></script>
  </body>
</html>
```

This template serves as the basic framework that includes the navbar and sidebar from separate files and provides a block for specific page content.

14.1.2 Separating the Navbar

The `navbar.html` template is a component that handles the top navigation section. This component will be called into `base.html`. By making it separate, we can reuse it across many pages without rewriting the code. Here is a simple example of the navbar:

```
<!-- File: apps/templates/dashboard/navbar.html -->
<nav class="navbar navbar-expand-lg navbar-light bg-
light">
    <div class="container-fluid">
        <a class="navbar-brand" href="{% url 'home'
%}">Platform Bot</a>
        <button class="navbar-toggler" type="button"
data-bs-toggle="collapse" data-bs-
target="#sidebarMenu" aria-controls="sidebarMenu"
aria-expanded="false" aria-label="Toggle sidebar">
            <span class="navbar-toggler-icon"></span>
        </button>
    </div>
</nav>
```

14.1.3 Separating the Sidebar

The sidebar is also a component that will be frequently used on the dashboard pages. Separating it into a different file allows us to make changes in a centralized manner. Here is an example for `sidebar.html`:

```
<!-- File: apps/templates/dashboard/sidebar.html -->
<div class="col-md-3">
    <div class="collapse d-md-block"
id="sidebarMenu">
        <nav id="sidebar" class="bg-light sidebar">
            <div class="position-sticky">
                <ul class="nav flex-column">
                    <li class="nav-item">
                        <a class="nav-link active"
href="{% url 'dashboard' %}">
                            <i class="bi bi-house-
door"></i> Dashboard
                        </a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{%
url 'profile' %}">
                            <i class="bi bi-
person"></i> Profile
                        </a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">
                            <i class="bi bi-
plus"></i> Create Bot
                        </a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">
                            <i class="bi bi-
gear"></i> Manage Bots
                        </a>
                    </li>
                    <!-- Logout Button in Sidebar -->
                    <li class="nav-item">
                        <a class="nav-link text-
danger" href="{% url 'logout' %}">
                            <i class="bi bi-box-
arrow-right"></i> Logout
                        </a>
                    </li>
                </ul>
            </div>
        </nav>
    </div>
</div>
```

14.1.4 Separating the Dashboard

The `dashboard.html` template is specific to the dashboard content, and this page will use `base.html` as its main framework. In this template, we will write content inside `{% block content %}`.

Here is an example:

```
<!-- File: apps/templates/dashboard/dashboard.html --
>
{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container">
    <div class="row">
        <div class="col-lg-12">
            <div class="jumbotron text-center bg-
primary text-white py-5">
                <h2>Welcome to the Dashboard,
{{ user_profile.user.username }}!</h2>
                <p>Here you can manage bots, view
statistics, and check your recent activity logs.</p>
            </div>
        </div>
    </div>
    <!-- Displaying messages if there are any errors
or successes -->
    {% if messages %}
        <div class="messages">
            {% for message in messages %}
                <div class="alert {{ message.tags
}}">{{ message }}</div>
            {% endfor %}
        </div>
    {% endif %}

    <div class="row mt-5">
        <div class="col-lg-12">
            <h3 class="text-center">Activity Log</h3>
            <table class="table table-striped table-
hover">
                <thead class="thead-dark">
                    <tr>
                        <th scope="col">Time</th>
                        <th scope="col">Activity</th>
                    </tr>
                </thead>
                <tbody>
                    {% for log in activity_logs %}
                    <tr>
                        <td>{{ log.timestamp|date:"d-
m-Y H:i" }}</td>
                        <td>{{ log.action }}</td>
                    </tr>
                    {% empty %}
                    <tr>
                        <td colspan="2" class="text-
center">No activity logs found.</td>
                    </tr>
                    {% endfor %}
                </tbody>
            </table>
        </div>
    </div>
</div>
{% endblock %}
```

By separating components like the navbar and sidebar into different files, we can keep the code more modular and easier to manage. Additionally, changes made to one component will automatically apply to all pages that use that component. A clear and clean template structure like this not only facilitates maintenance but also enhances efficiency when developing web-based applications.

14.2 Fixing The Pages

Next, we will start fixing all the dashboard pages, beginning with preparing the CSS and JS, dependencies, and others, to make the dashboard interface more interactive and responsive across various platforms.

14.2.1 Preparing CSS and JS for the Dashboard

Now we will start improving the dashboard's appearance. First, we will prepare the CSS and JS for the dashboard, and we will also discuss how the provided CSS and JavaScript code can be integrated into the Django project, explaining the function of each part of the code. Let's begin with the CSS for the sidebar layout and then see how JavaScript is used to enable charts and tables.

Preparing the CSS Code

The CSS file we are using is located in the following directory:

```
apps/static/assets/css/style.css
```

Here is the explanation of the provided CSS code.

```
:root {
    --offcanvas-width: 270px;
    --topNavbarHeight: 56px; /* Fixed from --
topNavbarMeight to --topNavbarHeight */
}
```

In this section, we define two global CSS variables that are used to determine the width of the sidebar (`--offcanvas-width`) and the height of the navbar (`--topNavbarHeight`). By using these variables, we can easily change the size of the sidebar or navbar throughout the application just by updating the variable values.

```
.sidebar-nav {
    width: var(--offcanvas-width);
}
```

The `.sidebar-nav` class is used to set the width of the sidebar according to the value of the variable we defined earlier (`--`

`offcanvas-width`). Here, the sidebar will have a fixed width of 270px.

```css
.sidebar-link {
    display: flex;
    align-items: center;
}
```

For `.sidebar-link`, we ensure that each link in the sidebar uses flexbox to arrange the elements inside it. The `align-items: center` ensures that the text and icons within the link are vertically centered.

```css
.sidebar-link .right-icon {
    display: inline-flex;
    transition: all ease 0.25s;
}

.sidebar-link[aria-expanded="true"] .right-icon {
    transform: rotate(180deg);
}
```

This is the setup for the dropdown icon in the sidebar. When a sidebar item is expanded (`aria-expanded="true"`), the icon will rotate 180 degrees with a smooth transition over 0.25 seconds.

CSS for Desktop (min-width: 992px)

```css
@media (min-width: 992px) {
    .offcanvas-start {
        transform: none !important;
        visibility: visible !important;
        position: fixed;
        top: var(--topNavbarHeight);
        height: calc(100% - var(--topNavbarHeight));
        z-index: 1030;
    }

    main {
        margin-left: var(--offcanvas-width);
        margin-top: var(--topNavbarHeight);
    }

    .navbar {
        z-index: 1040;
    }
}
```

For the desktop view, we ensure that the sidebar is always visible and not pushed aside. The sidebar is positioned fixed right below the navbar with a distance of `--topNavbarHeight`. The `main` also has a left margin to prevent the content from being obscured by the sidebar, and the navbar has a higher z-index to always remain above the sidebar.

CSS for Mobile (max-width: 991.98px)

```
@media (max-width: 991.98px) {
    .offcanvas-start {
        transform: translateX(-100%);
    }

    .offcanvas-start.show {
        transform: translateX(0);
    }
}
```

For the mobile view, the sidebar is set to be hidden by default (`transform: translateX(-100%)`). When the sidebar button is pressed, the `.show` class will be activated, and the sidebar will appear with a smooth transition.

Complete Code

```css
/* File: apps/static/assets/css/style.css */

:root {
    --offcanvas-width: 270px;
    --topNavbarHeight: 56px; /* Fixed from --
topNavbarMeight to --topNavbarHeight */
}

.sidebar-nav {
    width: var(--offcanvas-width);
}

.sidebar-link {
    display: flex;
    align-items: center;
}

.sidebar-link .right-icon {
    display: inline-flex;
    transition: all ease 0.25s;
}

.sidebar-link[aria-expanded="true"] .right-icon {
    transform: rotate(180deg);
}

@media (min-width: 992px) {
    /* Ensure sidebar always appears on desktop */
    .offcanvas-start {
        transform: none !important;
        visibility: visible !important;
        position: fixed;
        top: var(--topNavbarHeight);
        height: calc(100% - var(--topNavbarHeight));
        z-index: 1030; /* Ensure sidebar is above
other content */
    }
    /* Add margin for the main content so it's not
obscured by the sidebar */
    main {
        margin-left: var(--offcanvas-width);
        margin-top: var(--topNavbarHeight);
    }

    /* Ensure navbar is above the sidebar */
    .navbar {
        z-index: 1040; /* Higher than the sidebar */
    }
}

@media (max-width: 991.98px) {
    /* Hide sidebar on mobile */
    .offcanvas-start {
        transform: translateX(-100%);
    }

    .offcanvas-start.show {
        transform: translateX(0); /* Show sidebar
when button is pressed */
    }
}
```

Preparing the JavaScript Code

The JavaScript file used is located in the following directory:

```
apps/static/assets/js/script.js
```

In this section, JavaScript is used for two main functions: displaying charts using Chart.js and initializing DataTables for the data tables.

Activating Charts with Chart.js

```
const charts = document.querySelectorAll(".chart");
```

```
charts.forEach(function (chart) {
  var ctx = chart.getContext("2d");
  var myChart = new Chart(ctx, {
    type: "bar",
    data: {
      labels: ["Red", "Blue", "Yellow", "Green",
"Purple", "Orange"],
      datasets: [
        {
          label: "# of Votes",
          data: [12, 19, 3, 5, 2, 3],
          backgroundColor: [
            "rgba(255, 99, 132, 0.2)",
            "rgba(54, 162, 235, 0.2)",
            "rgba(255, 206, 86, 0.2)",
            "rgba(75, 192, 192, 0.2)",
            "rgba(153, 102, 255, 0.2)",
            "rgba(255, 159, 64, 0.2)",
          ],
          borderColor: [
            "rgba(255, 99, 132, 1)",
            "rgba(54, 162, 235, 1)",
            "rgba(255, 206, 86, 1)",
            "rgba(75, 192, 192, 1)",
            "rgba(153, 102, 255, 1)",
            "rgba(255, 159, 64, 1)",
          ],
          borderWidth: 1,
        },
      ],
    },
    options: {
      scales: {
        y: {
          beginAtZero: true,
        },
      },
    },
  });
});
```

In this code, we use Chart.js to create a chart.

1. `querySelectorAll(".chart")` searches for all elements with the chart class on the page.
2. Each chart element is looped through using `forEach`, and for each chart, we initialize the Chart using `getContext("2d")` to draw the chart within a 2D canvas.
3. Data for the bar chart is prepared with labels (color categories) and datasets (number of votes). We also add background colors and border colors for each bar in the chart.

Initializing DataTables

```
$(document).ready(function () {
  $(".data-table").each(function (_, table) {
    $(table).DataTable();
  });
});
```

In this section, we use the jQuery DataTables plugin to make the table more interactive. When the page is ready (`$(document).ready()`), each table with the data-table class will be initialized to be automatically sortable and filterable by DataTables.

Complete Code

```
// File: apps/static/assets/js/script.js

const charts = document.querySelectorAll(".chart");

charts.forEach(function (chart) {
  var ctx = chart.getContext("2d");
  var myChart = new Chart(ctx, {
    type: "bar",
    data: {
      labels: ["Red", "Blue", "Yellow", "Green",
"Purple", "Orange"],
      datasets: [
        {
          label: "# of Votes",
          data: [12, 19, 3, 5, 2, 3],
          backgroundColor: [
            "rgba(255, 99, 132, 0.2)",
            "rgba(54, 162, 235, 0.2)",
            "rgba(255, 206, 86, 0.2)",
            "rgba(75, 192, 192, 0.2)",
            "rgba(153, 102, 255, 0.2)",
            "rgba(255, 159, 64, 0.2)",
          ],
          borderColor: [
            "rgba(255, 99, 132, 1)",
            "rgba(54, 162, 235, 1)",
            "rgba(255, 206, 86, 1)",
            "rgba(75, 192, 192, 1)",
            "rgba(153, 102, 255, 1)",
            "rgba(255, 159, 64, 1)",
          ],
          borderWidth: 1,
        },
      ],
    },
    options: {
      scales: {
        y: {
          beginAtZero: true,
        },
      },
    },
  });
});

$(document).ready(function () {
  $(".data-table").each(function (_, table) {
    $(table).DataTable();
  });
```

```
});
```

This is the explanation for your CSS and JavaScript code. Make sure to include these files in your Django template so they can work properly on the dashboard page.

14.2.2 Improving the base.html Template for the Dashboard

Next, we will discuss how to improve the base.html template from its initial structure to make it more functional, with the addition of the necessary dependencies to manage various elements on the dashboard page. This code is an important part of the Django template setup, allowing us to organize layouts more efficiently and utilize various libraries, such as Bootstrap, DataTables, and Chart.js.

Improving the base.html Template

After enhancements, here is the upgraded code:

```
<!-- File: apps/templates/dashboard/base.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible"
content="IE=edge" />
    <meta name="viewport" content="width=device-
width, initial-scale=1.0" />
    <!-- Adding Bootstrap and Bootstrap Icons from
CDN -->
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-
beta3/dist/css/bootstrap.min.css" />
```

```
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.4.1/font/bootstrap-icons.css"/>
    <link rel="stylesheet"
href="https://cdn.datatables.net/1.10.24/css/dataTabl
es.bootstrap5.min.css" />
    <link rel="stylesheet" href="{{ ASSETS_ROOT
}}/css/style.css" />
    <title>{% block title %}Dashboard{% endblock
%}</title>
  </head>
  <body>
    <!-- Include Navbar -->
    {% include 'dashboard/navbar.html' %}

    <!-- Include Sidebar -->
    {% include 'dashboard/sidebar.html' %}

    <main class="mt-5 pt-3">
      <div class="container">
        {% block content %}
        <!-- Content will be injected here from other
templates -->
        {% endblock %}
      </div>
    </main>

    <!-- Adding JavaScript and jQuery CDN -->
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-
beta3/dist/js/bootstrap.bundle.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/chart.js@3.0.2/dist
/chart.min.js"></script>
    <script src="https://code.jquery.com/jquery-
3.5.1.js"></script>
    <script
src="https://cdn.datatables.net/1.10.24/js/jquery.dat
aTables.min.js"></script>
    <script
src="https://cdn.datatables.net/1.10.24/js/dataTables
.bootstrap5.min.js"></script>
```

```
    <script src="{{ ASSETS_ROOT
}}/js/script.js"></script>
  </body>
</html>
```

Let’s explain each part of these changes in more detail.

Adding Meta Tag for Compatibility

```
<meta http-equiv="X-UA-Compatible"
content="IE=edge" />
```

This tag ensures better compatibility in Internet Explorer by instructing the browser to use Edge mode, which supports modern HTML5 and CSS features.

Using Specific Versions of Bootstrap and DataTables

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-
beta3/dist/css/bootstrap.min.css" />
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap-
icons@1.4.1/font/bootstrap-icons.css"/>
<link rel="stylesheet"
href="https://cdn.datatables.net/1.10.24/css/dataTabl
es.bootstrap5.min.css" />
<link rel="stylesheet" href="{{ ASSETS_ROOT
}}/css/style.css" />
```

- ***Bootstrap v5.0.0-beta3***: This version is more stable and supports various modern UI components. We also include a consistent version of Bootstrap to avoid inconsistencies with other versions.

- ***DataTables with Bootstrap 5 Styling***: Added to ensure that interactive tables have a look that matches Bootstrap 5.
- ***Custom Styles***: The custom CSS file style.css from ASSETS_ROOT is included to allow for further custom styling.

Adding a Container for Main Content

```
<main class="mt-5 pt-3">
  <div class="container">
    {% block content %}
    <!-- Content will be injected here from other
templates -->
    {% endblock %}
  </div>
</main>
```

Adding the Bootstrap container class around the content block helps structure the content with more organized constraints within the Bootstrap grid layout.

Adding Various CDNs for JavaScript

```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-
beta3/dist/js/bootstrap.bundle.min.js"></script>
<script
src="https://cdn.jsdelivr.net/npm/chart.js@3.0.2/dist
/chart.min.js"></script>
<script src="https://code.jquery.com/jquery-
3.5.1.js"></script>
<script
src="https://cdn.datatables.net/1.10.24/js/jquery.dat
aTables.min.js"></script>
```

```
<script
src="https://cdn.datatables.net/1.10.24/js/dataTables
.bootstrap5.min.js"></script>
<script src="{{ ASSETS_ROOT
}}/js/script.js"></script>
```

The addition of the following libraries enables various enhanced features on the dashboard page:

- ***Bootstrap Bundle JS***: This includes Popper.js, which is used for dropdowns and other interactive components in Bootstrap.
- ***Chart.js***: This library is used for rendering charts on the dashboard.
- ***jQuery & DataTables***: jQuery is a required dependency for DataTables. DataTables is used to manage interactive tables that can be sorted, filtered, and paginated.
- ***Custom JavaScript***: The script.js file from ASSETS_ROOT will contain custom code, such as chart initialization or DataTables setup.

With these improvements, the base.html template is now more complete and ready for use in Django-based dashboard projects. This code includes:

1. The addition of Bootstrap and essential related dependencies for layout and styling.
2. Management of interactive tables using DataTables.
3. The addition of Chart.js to enable graphical data visualization.
4. Improved content structure with the use of containers and content modularization using blocks.

We now have a flexible template that supports modern dashboard development features.

14.2.3 Improving the Sidebar for the Dashboard

In this section, we will discuss the improvements made to the sidebar structure for the dashboard. The sidebar is an essential element of the dashboard as it serves as the main navigation that helps users access various features and pages within the application.

Previously, the sidebar utilized a fairly simple layout using Bootstrap's collapse class. However, to create a more responsive and modern sidebar appearance, we will enhance it by using the offcanvas component from Bootstrap, which is more flexible.

Issues with the Old Structure

1. ***Not Fully Responsive***: The sidebar used collapse, which might not be flexible enough for smaller screen size changes.
2. ***Rigid Layout***: By using the col-md-3 class, which forces the sidebar to fit into the Bootstrap grid, it made it difficult to use on various devices with small screens.
3. ***Lack of Interactivity***: There were not enough modern visual cues, such as dividers or more appealing headings, to separate the sidebar content.

To address these issues, we made changes using the offcanvas component from Bootstrap. Offcanvas provides the ability to display the sidebar with smoother animations and also supports a better user experience on smaller screens.

Below is the code after the improvements:

```
<!-- File: apps/templates/dashboard/sidebar.html -->
<div class="offcanvas offcanvas-start sidebar-nav bg-
dark" tabindex="-1" id="sidebar">
  <div class="offcanvas-body p-0">
    <nav class="navbar-dark">
      <ul class="navbar-nav">
        <li>
          <div class="text-muted small fw-bold text-
uppercase px-3">CORE</div>
        </li>
        <li>
          <a href="{% url 'dashboard' %}" class="nav-
link px-3 active">
            <span class="me-2"><i class="bi bi-
speedometer2"></i></span>
            <span>Dashboard</span>
          </a>
        </li>
        <li class="my-4"><hr class="dropdown-divider
bg-light"></li>
        <li>
          <div class="text-muted small fw-bold text-
uppercase px-3">BOTS</div>
        </li>
        <li>
          <a href="{% url 'create_bot' %}"
class="nav-link px-3">
            <span class="me-2"><i class="bi bi-plus-
circle-fill"></i></span>
            <span>Create Bot</span>
          </a>
        </li>
        <li>
          <a href="{% url 'manage_bots' %}"
class="nav-link px-3">
```

```
            <span class="me-2"><i class="bi bi-gear-
wide-connected"></i></span>
            <span>Manage Bots</span>
          </a>
        </li>
        <li class="my-4"><hr class="dropdown-divider
bg-light"></li>
        <li>
          <div class="text-muted small fw-bold text-
uppercase px-3 mb-3">Account</div>
        </li>
        <li>
          <a href="{% url 'logout' %}" class="nav-
link px-3 text-danger">
            <span class="me-2"><i class="bi bi-box-
arrow-right"></i></span>
            <span>Logout</span>
          </a>
        </li>
      </ul>
    </nav>
  </div>
</div>
```

Key Changes and Explanations

1. ***Using Offcanvas from Bootstrap***:
    - Offcanvas provides a more interactive user experience, especially for smaller screens. It allows the sidebar to be displayed with a smooth animation from the side.
    - `tabindex="-1"` is used to ensure that the sidebar does not gain focus automatically when loaded.
    - By using `offcanvas-body`, we can ensure that the sidebar content will be responsive and will not affect the layout of the main page.
2. ***More Modern Appearance***:

- ***Divider***: Added `<hr class="dropdown-divider bg-light">` to provide a visual separator between navigation items, helping users better understand the sidebar structure.
- ***Small Headings***: Added small heading text with the classes `text-muted small fw-bold text-uppercase px-3`, providing a clearer division between categories in the sidebar.

3. ***More Descriptive Icons***:
    - Using more descriptive icons for each link, such as `bi-speedometer2` for the dashboard and `bi-plus-circle-fill` for the "Create Bot" button, makes the sidebar more informative.
4. ***Logout Interaction***:
    - The logout button is given red text using the `text-danger` class to make it stand out and be easily found by users.

With these improvements, the sidebar is not only more responsive and user-friendly on small screens, but it also provides a more modern and professional appearance. The more organized structure and the use of the offcanvas component from Bootstrap make the sidebar more flexible for various screen sizes without compromising its functionality on desktop or mobile.

14.2.4 Fixing the Navbar for the Dashboard

Now we will continue by fixing the navbar that we have separated into the `navbar.html` file. We will change the structure of the navbar to make it more modern and functional.

This navbar uses Bootstrap 5 and includes several features such as an off-canvas sidebar, a search form, and a profile dropdown. Additionally, we will store the logo used in the navbar in the folder `apps/static/assets/img/`.

Let’s take a look at the new navbar structure:

```
<!-- File: templates/navbar.html -->
<!-- Navbar -->
<nav class="navbar navbar-expand-lg navbar-dark bg-
dark fixed-top">
  <div class="container-fluid">
    <!-- Toggler Button for Off-Canvas Sidebar -->
    <button class="navbar-toggler" type="button"
data-bs-toggle="offcanvas" data-bs-target="#sidebar"
aria-controls="sidebar">
      <span class="navbar-toggler-icon"></span>
    </button>

    <!-- Logo and Navbar Text -->
    <a class="navbar-brand me-auto ms-lg-0 ms-3 text-
uppercase fw-bold" href="{% url 'home' %}">
      <img src="{{ ASSETS_ROOT }}/img/logo.png"
alt="" width="30" height="24" class="d-inline-block
align-text-top"> Platform Bot
    </a>

    <!-- Toggler Button for Navbar -->
    <button class="navbar-toggler" type="button"
data-bs-toggle="collapse" data-bs-target="#topNavBar"
aria-controls="topNavBar" aria-expanded="false" aria-
label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>

    <!-- Collapsible Navbar Section -->
    <div class="collapse navbar-collapse"
id="topNavBar">
      <!-- Search Form -->
      <form class="d-flex ms-auto my-3 my-lg-0">
```

```
        <div class="input-group">
          <input class="form-control" type="search"
placeholder="Search" aria-label="Search" />
          <button class="btn btn-primary"
type="submit">
            <i class="bi bi-search"></i>
          </button>
        </div>
      </form>

      <!-- Profile Dropdown -->
      <ul class="navbar-nav">
        <li class="nav-item dropdown">
          <a class="nav-link dropdown-toggle ms-2"
href="#" role="button" data-bs-toggle="dropdown"
aria-expanded="false">
            <i class="bi bi-person-fill"></i>
          </a>
          <ul class="dropdown-menu dropdown-menu-
end">
            <li><a class="dropdown-item" href="{% url
'profile' %}">Profile</a></li>
          </ul>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

This navbar is built with several important elements:

1. ***Toggler Button for Off-Canvas Sidebar:*** Controls the sidebar to appear from the side.
2. ***Logo:*** Displays our logo taken from the `img` folder.
3. ***Toggler Button for Navbar:*** Used to show or hide the navbar section on smaller devices.
4. ***Search Form:*** Located on the right side of the navbar for easy searching.

5. ***Profile Dropdown:*** Contains a menu to access the user profile and other account-related features.

Saving the Logo

To display the logo in the navbar, we need to save the logo image file in the `img` folder within the static folder structure. We can do this by placing our logo in:

```
apps/
    static/
        assets/
            img/
                logo.png
```

With this, when we use `{{ ASSETS_ROOT }}/img/logo.png` in the template, our logo image will be fetched from the `static/assets/img/logo.png` folder.

Explanation of Logo Placement

In the `navbar.html` template, the logo is defined with the following `<img>` tag:

```
<img src="{{ ASSETS_ROOT }}/img/logo.png" alt=""
width="30" height="24" class="d-inline-block align-
text-top">
```

This tag ensures the logo image is displayed at a size of 30x24 pixels in the navbar. The `src` attribute refers to the location of the image in the static folder, and `class="d-inline-block align-text-top"` ensures that the logo is vertically aligned with the text next to it.

By separating the navbar into the `navbar.html` file and storing the logo in the `img` folder, we have improved the organization and modularity of our code structure. This will make it easier to maintain the project in the future and allow us to develop new features more easily.

14.2.5 Improving the Dashboard Page

In this section, we will discuss the shortcomings of the previous dashboard structure, the changes that have been made, and the addition of new elements to enrich the user experience. We will also discuss the **charts** section, which still requires integration to function.

Shortcomings of the Previous Dashboard Code

1. ***Lack of Dynamic and Modern Appearance:***
    - Previously, the dashboard displayed static elements like activity log tables and status messages. However, there were no visually appealing elements like **cards** to provide an overview of the data.
    - The **jumbotron** at the top used a design without shadow effects or rounded corners, making it look a bit rigid.
2. ***Lack of Comprehensive Visual Information:***
    - The previous dashboard did not provide a good way to present information concisely through data visualizations such as **charts** or **cards**.
    - Users were only shown an activity log table without broader context, making it difficult for

users to understand the system's status or progress.

3. ***Unintegrated Error and Success Messages:***
   - Error or success messages were only displayed in a simple format, making them less noticeable, especially on a busy dashboard.

Below is more complete code to modernize the dashboard, complete with cards, more harmonious status messages, and placeholders for **charts**. This also includes basic integration for error or success messages and provides a template for charts, ready to be connected with data in the future.

```
<!-- File: apps/templates/dashboard/dashboard.html --
>

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="row mb-4">
  <div class="col-md-12 fw-bold fs-3">
    Dashboard
  </div>
</div>

<!-- Jumbotron Welcome Section -->
<div class="row mb-4">
  <div class="col-lg-12">
    <div class="jumbotron text-center bg-primary
text-white py-5 shadow-sm rounded">
      <h2>Welcome to the Dashboard,
{{ user_profile.user.username }}!</h2>
      <p>Here you can manage bots, view statistics,
and check your latest activity logs.</p>
    </div>
  </div>
</div>
```

```
<!-- Cards Section -->
<div class="row">
  <div class="col-md-3 mb-3">
    <div class="card bg-primary text-white h-100
shadow-sm">
      <div class="card-body d-flex align-items-center
justify-content-between">
        <span class="fs-5">Primary Card</span>
        <i class="bi bi-bar-chart-line fs-3"></i>
      </div>
      <div class="card-footer d-flex align-items-
center">
        <span>View Details</span>
        <span class="ms-auto">
          <i class="bi bi-chevron-right"></i>
        </span>
      </div>
    </div>
  </div>

  <div class="col-md-3 mb-3">
    <div class="card bg-warning text-dark h-100
shadow-sm">
      <div class="card-body d-flex align-items-center
justify-content-between">
        <span class="fs-5">Warning Card</span>
        <i class="bi bi-exclamation-triangle-fill fs-
3"></i>
      </div>
      <div class="card-footer d-flex align-items-
center">
        <span>View Details</span>
        <span class="ms-auto">
          <i class="bi bi-chevron-right"></i>
        </span>
      </div>
    </div>
  </div>

  <div class="col-md-3 mb-3">
    <div class="card bg-success text-white h-100
shadow-sm">
```

```
      <div class="card-body d-flex align-items-center
justify-content-between">
        <span class="fs-5">Success Card</span>
        <i class="bi bi-check-circle-fill fs-3"></i>
      </div>
      <div class="card-footer d-flex align-items-
center">
        <span>View Details</span>
        <span class="ms-auto">
          <i class="bi bi-chevron-right"></i>
        </span>
      </div>
    </div>
  </div>

  <div class="col-md-3 mb-3">
    <div class="card bg-danger text-white h-100
shadow-sm">
      <div class="card-body d-flex align-items-center
justify-content-between">
        <span class="fs-5">Danger Card</span>
        <i class="bi bi-x-circle-fill fs-3"></i>
      </div>
      <div class="card-footer d-flex align-items-
center">
        <span>View Details</span>
        <span class="ms-auto">
          <i class="bi bi-chevron-right"></i>
        </span>
      </div>
    </div>
  </div>
</div>

<!-- Activity Log Section -->
<div class="row">
  <div class="col-md-12 mb-3">
    <div class="card shadow-sm">
      <div class="card-header d-flex align-items-
center">
        <span><i class="bi bi-clock-history me-2
text-primary"></i></span>
        <span class="fw-bold">Activity Log</span>
      </div>
      <div class="card-body">
        <div class="table-responsive">
          <table id="activity-log-table" class="table
table-striped table-hover data-table" style="width:
100%">
            <thead>
              <tr>
                <th>Time</th>
                <th>Activity</th>
              </tr>
            </thead>
            <tbody>
              {% for log in activity_logs %}
              <tr>
                <td>{{ log.timestamp|date:"d-m-Y H:i"
}}</td>
                <td>{{ log.action }}</td>
              </tr>
              {% empty %}
              <tr>
                <td colspan="2" class="text-
center">No activity logs found.</td>
              </tr>
              {% endfor %}
            </tbody>
          </table>
        </div>
      </div>
    </div>
  </div>
</div>

<!-- Placeholder for Charts -->
<div class="row">
  <div class="col-md-6 mb-3">
    <div class="card h-100 shadow-sm">
      <div class="card-header d-flex align-items-
center">
        <span class="me-2"><i class="bi bi-bar-chart-
fill text-primary"></i></span>
        <span class="fw-bold">Activity Chart</span>
      </div>
      <div class="card-body">
        <canvas class="chart" width="400"
height="200" id="activity-chart"></canvas>
      </div>
    </div>
  </div>

  <div class="col-md-6 mb-3">
    <div class="card h-100 shadow-sm">
      <div class="card-header d-flex align-items-
center">
        <span class="me-2"><i class="bi bi-bar-chart-
fill text-primary"></i></span>
        <span class="fw-bold">Log Activity</span>
      </div>
      <div class="card-body">
        <canvas class="chart" width="400"
height="200" id="log-chart"></canvas>
      </div>
    </div>
  </div>
</div>

{% endblock %}
```

To link these **charts** with data, we need to use JavaScript and a chart library like **Chart.js**. This can be integrated in the next phase according to the data you want to display.

Key Changes Made to the Dashboard

In this dashboard improvement, we have added several important components such as **cards**, **more engaging status messages**, and **chart placeholders**. Although the charts are not yet integrated with data, these placeholders provide a better visual representation of the potential data to be displayed in the future.

1. ***Addition of Cards:***

- Cards are used to display important information quickly and visually. Each card has an icon and a footer that adds interactivity. For example, there is a **primary card** to display primary data and a **success card** to show success data.
- These cards give the dashboard a more modern and professional feel, while also helping users grasp the data at a glance.

2. ***More Cohesive Error and Success Messages:***

- Success and error messages now have a more attractive appearance and are well integrated. Each message is displayed using alerts with **tags** to clarify the status of the message, such as success or error. This helps users quickly identify important messages.
- The visualization of the messages is now more **harmonious with the overall design of the dashboard**, ensuring consistency and a better experience.

3. ***More Responsive Activity Log Table:***

- The activity log table is still displayed, but it is now equipped with a more **prominent header** and is shown within a card component, making its appearance more cohesive with other elements on the dashboard.
- This table has also been made **responsive** to ensure it looks good on various screen sizes.

4. ***Placeholders for Charts:***

- Although the charts are currently not integrated with any data, two canvas placeholders have been prepared to **display activity** and l**og activity graphs**.

- The addition of these charts is intended to provide **space for future data visualization**, which will enrich the information that can be presented on the dashboard.
- With these **placeholders**, we can begin planning the integration of data using libraries like **Chart.js** or **ApexCharts** in the future.

Integrating Charts

To integrate charts in the future, the following steps can be taken:

1. ***Choose the Right Chart Library:***
    - Use Chart.js or ApexCharts to add data visualization to the dashboard. Both libraries support various types of graphs such as bar charts, line charts, and pie charts, which are very flexible for displaying data.
2. ***Connect Charts with Data:***
    - After the library is added, connect the charts with backend data (such as user statistics, bot performance, or activity logs) retrieved from the database. This can be done through AJAX or APIs prepared on the Django server.
3. ***Interactivity and Data Filtering:***
    - If possible, add interactive features such as time filters or category selections to give users more control over the data they want to see.

With this updated design, the dashboard becomes more professional, modern, and ready to accommodate more visual data that can help users manage their activities on the platform. Clearer and more engaging messages, along with the use of interactive elements like cards and charts, significantly enhance the user experience compared to before.

14.3 Improvement Of The Profile Page

In updating the profile page `profile.html`, the main goal is to create a more professional and attractive appearance. First, we expanded the use of Bootstrap components to ensure a responsive layout and better aesthetics. This was achieved by adding icons from Bootstrap Icons for each label, providing more informative and appealing visuals for users.

Here is the revised code:

```
<!-- apps/templates/users/profile.html -->
{% extends "dashboard/base.html" %}

{% block content %}
<div class="content">
    <div class="row">
        <div class="col-md-8">
            <div class="card bg-light shadow-sm">
                <div class="card-header bg-primary
text-white">
                    <h5 class="title">
                        <i class="bi bi-pencil-
square"></i> Edit Profile
                    </h5>
                </div>
                <div class="card-body">
```

```
                    <form method="post"
enctype="multipart/form-data">
                        {% csrf_token %}

                        <!-- Username and Email -->
                        <div class="row mb-3">
                            <div class="col-md-6">
                                <div class="form-
group">
                                    <label class="fw-
bold"><i class="bi bi-person-fill"></i>
Username</label>
                                    {{ form.username
}}
                                </div>
                            </div>
                            <div class="col-md-6">
                                <div class="form-
group">
                                    <label class="fw-
bold"><i class="bi bi-envelope-fill"></i>
Email</label>
                                    {{ form.email }}
                                </div>
                            </div>
                        </div>

                        <!-- First Name and Last Name
-->
                        <div class="row mb-3">
                            <div class="col-md-6">
                                <div class="form-
group">
                                    <label class="fw-
bold"><i class="bi bi-file-person"></i> First
Name</label>

{{ form.first_name }}
                                </div>
                            </div>
                            <div class="col-md-6">
                                <div class="form-
group">
```

```
                                    <label class="fw-
bold"><i class="bi bi-file-person-fill"></i> Last
Name</label>
                                    {{ form.last_name
}}
                                </div>
                            </div>
                        </div>

                        <!-- Company -->
                        <div class="form-group mb-3">
                            <label class="fw-bold"><i
class="bi bi-building"></i> Company</label>
                            {{ form.company }}
                        </div>

                        <!-- Address -->
                        <div class="form-group mb-3">
                            <label class="fw-bold"><i
class="bi bi-house"></i> Address</label>
                            {{ form.address }}
                        </div>

                        <!-- City, Country, Postal
Code -->
                        <div class="row mb-3">
                            <div class="col-md-4">
                                <div class="form-
group">
                                    <label class="fw-
bold"><i class="bi bi-geo-alt-fill"></i> City</label>
                                    {{ form.city }}
                                </div>
                            </div>
                            <div class="col-md-4">
                                <div class="form-
group">
                                    <label class="fw-
bold"><i class="bi bi-globe2"></i> Country</label>
                                    {{ form.country
}}
                                </div>
                            </div>
                            <div class="col-md-4">
                                <div class="form-
group">
                                    <label class="fw-
bold"><i class="bi bi-envelope"></i> Postal
Code</label>

{{ form.postal_code }}
                                </div>
                            </div>
                        </div>

                        <!-- About Me -->
                        <div class="form-group mb-3">
                            <label class="fw-bold"><i
class="bi bi-info-circle"></i> About Me</label>
                            {{ form.about }}
                        </div>

                        <!-- Profile Picture -->
                        <div class="form-group mb-3">
                            <label class="fw-bold"><i
class="bi bi-camera"></i> Profile Picture</label>
                            {{ form.foto_profile }}
                        </div>

                        <!-- Submit Button -->
                        <button type="submit"
class="btn btn-fill btn-success">
                            <i class="bi bi-
save"></i> Save
                        </button>
                    </form>
                </div>
            </div>
        </div>

        <!-- Profile Preview Section -->
        <div class="col-md-4">
            <div class="card card-user text-center
shadow-sm">
                <div class="card-body">
                    <div class="author">
```

```
                        <img class="avatar img-fluid
rounded-circle" src="{% if form.foto_profile.value %}
{{ form.foto_profile.value.url }}{% else
%}/static/assets/img/default-avatar.png{% endif %}"
alt="Profile Picture">
                        <h5 class="title text-
center">{{ form.username.value }}</h5>
                    </div>
                    <div class="card-description
text-center">
                        {{ form.about.value }}
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

The first aspect to notice is the structure of the form. The form is organized so that each label is positioned above the corresponding input field, using the Bootstrap class `form-group`. This not only creates consistent spacing but also enhances readability. Each label is marked with the `fw-bold` class, making it more prominent and easier for users to identify each field that needs to be filled out.

For the input fields, the `form-control` class is applied to all input elements, including `TextInput`, `EmailInput`, and `Textarea`. This provides a uniform appearance and improves the user experience when interacting with the form. Placeholders are also added to provide further guidance on the information that needs to be entered.

Next, the layout of the form is well organized into several rows. For example, the fields for username and email are grouped in one row, while the fields for first name and last name are grouped in the next row. This not only creates a clearer structure but also maximizes the use of screen space, especially on smaller devices.

In addition, we added icons for each label, which serve as additional visual elements that attract the user's attention. The icons provide further context about the information being requested, such as a house icon for the address and a globe icon for the country. This addition makes the form more intuitive and helps users understand the purpose of each field more easily.

Finally, for the save button, we assigned the `btn-success` class to highlight the button with a bright and appealing color, inviting users to press the button after filling out the form. An icon is included in the button to enhance visual appeal.

With all these changes, the profile page `profile.html` is now not only more visually appealing but also more functional and user-friendly, creating a better experience for users in editing their profiles.

Chapter Conclusion

In this chapter, we discussed the enhancement of the dashboard page in the Django project, focusing on improving the structure, appearance, and functionality of the dashboard. Here is a summary of each subsection:

1. ***Separating Template Directory Structure***
    - ***Creating base.html Template***: The structure of the base.html template is designed to serve as the foundation for the entire dashboard layout. This template establishes the basic framework, including elements such as the header, footer, and space for dynamic content.
    - ***Creating navbar.html Template***: The navbar.html template contains the code for the navbar, which is the main navigation element in the dashboard. This navbar ensures that navigation between different sections of the dashboard is easy and structured.
    - ***Creating sidebar.html Template***: The sidebar.html template organizes the sidebar that serves as additional navigation for the dashboard features. This sidebar is designed to display frequently used menu options.
    - ***Creating dashboard.html Template***: The dashboard.html template is where the specific content of the dashboard page is displayed. This template integrates various components and provides space to present key data and information.

2. ***Improving the Page***
    - ***Setting Up CSS and JS for the Dashboard***: The CSS and JavaScript code is configured to enhance the functionality and design of the dashboard. This involves adjusting styles and scripts to support elements such as charts, tables, and dynamic interactions.
    - ***Improving base.html Template for the Dashboard***: The base.html template has been updated to reflect a better design and structure. Changes include adding links to stylesheets and scripts that support the dashboard's appearance and functionality.
    - ***Improving the Sidebar for the Dashboard***: The sidebar has been updated to ensure its appearance aligns with the dashboard theme and provides efficient navigation. These changes include rearranging components and adjusting styles for better consistency.
    - ***Improving the Navbar for the Dashboard***: The navbar has been adjusted to fit the design and specific needs of the dashboard. These changes ensure that the navbar functions well and has a consistent appearance.
    - ***Enhancing the Dashboard Appearance***: The dashboard's appearance has been enhanced by adding visual components such as cards and charts, as well as improving the design to be more attractive and functional. This also includes

testing to ensure that the appearance is responsive and user-friendly.

By following these steps, the dashboard page can be thoroughly enhanced, offering a better user experience and a more modern design. The focus on separating the template structure and improving individual elements ensures that the dashboard not only looks professional but is also easy to use and dynamic.

Chapter 15 - Refining the Bots Page

In this section, we will focus on enhancing the bot-related pages in the Django application, which include pages for creating, managing, and editing bots, as well as adding and managing commands.

The main focus is to improve the interface and functionality of the following pages:

- ***Create Bot Page:*** A place where users can create a new bot by filling out a form.
- ***Manage Bots Page:*** A place where users can view a list of created bots, manage them, and perform actions such as editing or deleting a bot.
- ***Edit Bot Page:*** A place where users can update the information of an existing bot.
- ***Add Command Page:*** A place where users can add new commands to the bot.
- ***Edit Command Page:*** A place where users can edit existing commands.

For each page, we will focus on improving the design and functionality by utilizing Bootstrap components. These modifications aim to provide a more professional appearance and a better user experience, without the need for additional CSS writing.

Here is the structure that will be discussed and refined in this chapter:

```
apps/
│
└── templates/
    └── bots/
        ├── create_bot.html
        ├── manage_bots.html
        ├── edit_bot.html
        ├── add_command.html
        └── edit_command.html
```

Let’s begin with the discussion and improvement of each page.

15.1 Improving The Create Bot Page

The Create Bot page is a crucial feature in the bot_platform application, allowing users to create new bots by entering a name, token, and selecting the bot platform. The design of this page should be responsive, accessible, and provide a good user experience. To enhance this page, we will implement several significant changes that leverage Bootstrap components and add elements to handle error messages.

Design Enhancements for the Page

In the previous version, the form on this page already utilized several Bootstrap components. However, we can improve its appearance by adding new elements such as icons, colors, and a more interactive design. Additionally, we will add a modal to display error messages if they occur.

Changes to the Code

1. ***Addition of Icons and Colors:***
   - Add icons to the form labels to provide better visual context.
   - Use Bootstrap colors for the form borders and background.
2. ***Enhancement of Form Appearance:***
   - Change the submit button to a large button with an icon to improve visibility.
   - Add placeholders to the input fields to provide additional guidance to users.
3. ***Error Message Modal:***
   - Add a Bootstrap modal to display error messages if there is a problem submitting the form.
   - This modal will automatically appear if there are errors in the form.

Complete Code After Refinement

```
<!-- File: apps/templates/bots/create_bot.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <!-- Card for the Form -->
    <div class="card shadow-lg" style="border-radius:
15px;">
        <div class="card-header text-center bg-
primary text-white" style="border-radius: 15px 15px 0
0;">
            <h5 class="card-title"><i class="bi bi-
bot-fill"></i> Create New Bot</h5>
        </div>
        <div class="card-body">
            <!-- Form for the Bot -->
            <form method="post">
                {% csrf_token %}

                <!-- Form Group for Platform (appears
first) -->
                <div class="form-group mb-4">
                    <label class="form-label fw-
bold">
                        <i class="bi bi-device-
screenshot"></i> Platform
                    </label>
                    <div class="btn-group d-flex
justify-content-center" role="group" aria-
label="Platform">
                        <input type="radio"
name="bot_type" id="telegram" value="telegram" {% if
form.bot_type.value == 'telegram' %}checked{% endif
%} onclick="toggleFields()" class="btn-check">
                        <label class="btn btn-
outline-primary" for="telegram"><i class="bi bi-
telegram"></i> Telegram</label>

                        <input type="radio"
name="bot_type" id="whatsapp" value="whatsapp" {% if
form.bot_type.value == 'whatsapp' %}checked{% endif
%} onclick="toggleFields()" class="btn-check">
                        <label class="btn btn-
outline-primary" for="whatsapp"><i class="bi bi-
whatsapp"></i> WhatsApp</label>
                    </div>
                </div>

                <!-- Form Group for Name -->
                <div class="form-group mb-4">
                    <label
for="{{ form.name.id_for_label }}" class="fw-bold">
                        <i class="bi bi-chat-
dots"></i> Name
                    </label>
                    <input type="text" name="name"
id="{{ form.name.id_for_label }}"
```

```
value="{{ form.name.value }}" class="form-control {%
if form.name.errors %}is-invalid{% endif %}">
                    {% for error in form.name.errors
%}
                        <div class="invalid-
feedback">{{ error }}</div>
                    {% endfor %}
                </div>

                <!-- Form Group for Token (only for
Telegram) -->
                <div class="form-group mb-4"
id="token-field">
                    <label
for="{{ form.token_telegram.id_for_label }}"
class="fw-bold">
                        <i class="bi bi-lock-
fill"></i> Token
                    </label>
                    <input type="text"
name="token_telegram"
id="{{ form.token_telegram.id_for_label }}" value="{{
form.token_telegram.value }}" class="form-control {%
if form.token_telegram.errors %}is-invalid{% endif
%}">
                    {% for error in
form.token_telegram.errors %}
                        <div class="invalid-
feedback">{{ error }}</div>
                    {% endfor %}
                </div>

                <!-- Submit Button -->
                <div class="text-center">
                    <button type="submit" class="btn
btn-primary btn-lg">
                        <i class="bi bi-bot-
fill"></i> Create Bot
                    </button>
                </div>
            </form>
        </div>
    </div>
    <!-- Modal for Error Messages -->
    {% if form.errors %}
    <div class="modal fade" id="errorModal"
tabindex="-1" aria-labelledby="errorModalLabel" aria-
hidden="true">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header bg-danger
text-white">
                    <h1 class="modal-title fs-5"
id="errorModalLabel">An Error Occurred</h1>
                    <button type="button" class="btn-
close" data-bs-dismiss="modal" aria-
label="Close"></button>
                </div>
                <div class="modal-body">
                    <div class="alert alert-danger">
                        {% for field in form %}
                            {% for error in
field.errors %}
                                <p>{{ error }}</p>
                            {% endfor %}
                        {% endfor %}
                        {% for error in
form.non_field_errors %}
                            <p>{{ error }}</p>
                        {% endfor %}
                    </div>
                </div>
                <div class="modal-footer">
                    <button type="button" class="btn
btn-secondary" data-bs-dismiss="modal">Close</button>
                </div>
            </div>
        </div>
    </div>

    <script>
        // Ensure that Bootstrap JavaScript is loaded
and jQuery if needed
        document.addEventListener('DOMContentLoaded',
function () {
```

```
            if
(document.querySelector('#errorModal')) {
                var errorModal = new
bootstrap.Modal(document.getElementById('errorModal')
);
                errorModal.show();
            }
        });
    </script>
    {% endif %}
</div>

<script>
    // Function to show/hide elements based on
platform
    function toggleFields() {
        var tokenField =
document.getElementById('token-field');
        var whatsappFields =
document.getElementById('whatsapp-fields');

        if
(document.getElementById('telegram').checked) {
            tokenField.classList.remove('d-none');
            whatsappFields?.classList.add('d-none');
        } else if
(document.getElementById('whatsapp').checked) {
            tokenField.classList.add('d-none');
            whatsappFields?.classList.remove('d-
none');
        }
    }

    // Call toggleFields() when the page loads
    document.addEventListener('DOMContentLoaded',
function () {
        toggleFields();
    });
</script>
{% endblock %}
```

Explanation of Changes:

- ***Card Header:*** Added a bot icon to the card title for a more visually appealing effect.
- ***Platform Selection:*** Used stylish radio buttons with icons for the Telegram and WhatsApp platforms.
- ***Labels and Inputs:*** Each label now has a corresponding icon to enhance visual appeal.
- ***Submit Button:*** An icon was added to the button to make it more informative and attractive.
- ***Error Modal:*** Remains as it was, but with consistent style and function.

With these improvements, the appearance of the `create_bot.html` page will be more professional, modern, and appealing.

Shortcomings of the Previous Code

The previous code already used basic Bootstrap components but did not fully utilize the design capabilities offered by Bootstrap, such as icons and colors that could enhance the user experience. Additionally, there was no handling for displaying error messages, which is crucial for providing feedback to users if the form submission fails. The addition of a modal for error messages helps ensure that users are informed about issues in a clear and structured manner.

15.2 Fixing The Manage Bots Page

The Manage Bots page gives users the ability to view and manage the list of existing bots. This page must be designed to be easy to navigate and provide clear information about the status and actions that can be taken on each bot.

Enhancements to Page Design

In previous versions, this page used a simple table to display bot data. While functional enough, this layout could be further improved to enhance the user experience by adding several additional design elements and refining the table appearance.

Changes to the Code

1. ***Addition of Cards and Header:***
    - Utilizing Bootstrap card components to provide a more attractive and structured design.
    - Adding icons to the card header to provide better visual context.
2. ***Responsive Table and Hover Effects:***
    - Using `table-responsive` to ensure the table adapts to various screen sizes.
    - Adding the `table-hover` class to provide a hover effect on table rows, improving readability and user interaction.
3. ***Icons and Badges in Platform and Status Columns:***
    - Using icons to indicate the bot platform with appropriate colors.

- Utilizing Bootstrap badges to display the bot's status (Active/Inactive) with different colors.

4. ***More Informative Edit and Delete Buttons:***
   - Modifying the Edit and Delete buttons to include appropriate icons, enhancing visibility and understanding of the button functions.

Complete Code After Refinement

```
<!-- File: apps/templates/bots/manage_bots.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container my-5">
    <!-- Header Section -->
    <div class="bg-primary text-white text-center p-4
rounded mb-4">
        <h1 class="display-6"><i class="bi bi-
robot"></i> Manage Your Bots</h1>
        <p class="lead">Easily manage, edit, and
monitor your Telegram and WhatsApp bots</p>
    </div>
<!-- Displaying error or success messages -->
{% if messages %}
  <div class="messages">
    {% for message in messages %}
      <div class="alert {{ message.tags }} shadow-sm
rounded">
        {{ message }}
      </div>
    {% endfor %}
  </div>
{% endif %}

    <!-- Bots Cards -->
    <div class="row">
        {% for bot in bots %}
        <div class="col-md-6 col-lg-4 mb-4">
```

```
            <div class="card shadow-sm h-100">
                <div class="card-header bg-primary
text-white d-flex justify-content-between align-
items-center">
                    <h5 class="mb-0">
                        <i class="bi bi-robot"></i>
{{ bot.name }}
                    </h5>
                    <span class="badge
{{ bot.is_active|yesno:'bg-success,bg-danger' }}">
                        {{ bot.is_active|
yesno:'Active,Inactive' }}
                    </span>
                </div>

                <div class="card-body">
                    <!-- Platform -->
                    <p><strong>Platform:</strong>
                        {% if bot.bot_type ==
'telegram' %}
                        <i class="bi bi-telegram
text-info"></i> Telegram
                        {% elif bot.bot_type ==
'whatsapp' %}
                        <i class="bi bi-whatsapp
text-success"></i> WhatsApp
                        {% endif %}
                    </p>

                    <!-- Credentials (only for
Telegram) -->
                    {% if bot.bot_type == 'telegram'
%}

<p><strong>Credentials:</strong></p>
                    <ul class="list-unstyled">
                        <li><i class="bi bi-key-
fill"></i> <strong>Token:</strong>
{{ bot.token_telegram }}</li>
                    </ul>
                    {% endif %}

                    <!-- Commands -->
                    <p><strong>Commands:</strong></p>
                    <table class="table table-sm
table-bordered">
                        <thead>
                            <tr>
                                <th>Command</th>
                                <th>Response</th>
                                <th>Actions</th>
                            </tr>
                        </thead>
                        <tbody>
                            {% for command in
bot.commands.all %}
                            <tr>

<td><strong>{{ command.command }}</strong></td>

<td>{{ command.response }}</td>
                                <td>
                                    <div class="d-
flex gap-2">
                                        <a href="{%
url 'edit_command' command.id %}" class="btn btn-
warning btn-sm">
                                            <i
class="bi bi-pencil"></i> Edit
                                        </a>
                                        <form
method="post" style="display:inline;">
                                            {%
csrf_token %}
                                            <button
type="submit" name="delete_command"
value="{{ command.id }}" class="btn btn-danger btn-
sm" onclick="return confirm('Are you sure you want to
delete this command?');">
                                                <i
class="bi bi-trash"></i> Delete
                                            </button>
                                        </form>
                                    </div>
                                </td>
                            </tr>
                            {% empty %}
                            <tr>
                                <td colspan="3"
class="text-center">No commands available.</td>
                            </tr>
                            {% endfor %}
                        </tbody>
                    </table>

                    <!-- Add Command Button -->
                    <a href="{% url 'add_command'
bot.id %}" class="btn btn-success btn-sm mt-2">
                        <i class="bi bi-plus-
circle"></i> Add Command
                    </a>
                </div>

                <!-- Card Footer: Actions -->
                <div class="card-footer bg-light">
                    <div class="d-flex justify-
content-between">
                        <a href="{% url 'edit_bot'
bot.id %}" class="btn btn-primary btn-sm">
                            <i class="bi bi-
pencil"></i> Edit Bot
                        </a>
                        <form method="post"
style="display:inline;">
                            {% csrf_token %}
                            <button type="submit"
name="delete" value="{{ bot.id }}" class="btn btn-
danger btn-sm" onclick="return confirm('Are you sure
you want to delete this bot?');">
                                <i class="bi bi-
trash"></i> Delete
                            </button>
                        </form>
                    </div>
                </div>
            </div>
        </div>
        {% empty %}
        <div class="col-12">
```

```
            <div class="alert alert-warning text-
center" role="alert">
                No bots available.
            </div>
        </div>
        {% endfor %}
    </div>

    <!-- Button to Create New Bot -->
    <div class="text-center mt-4">
        <a href="{% url 'create_bot' %}" class="btn
btn-lg btn-primary">
            <i class="bi bi-plus-circle"></i> Create
New Bot </a>
 </div>
</div>
{% endblock %}
```

Explanation of Improvements:

1. ***Header Section:*** Emphasis added using the icon `<i class="bi bi-robot"></i>` to strengthen the bot management theme.
2. ***More Informative Cards:*** Each bot is displayed in a card with clear information, including platform and active/inactive status. WhatsApp does not have credentials (as desired), so only Telegram displays the token.
3. ***Command Table:*** The structure of the commands table is more organized with clearer headers and the use of Bootstrap icons for edit and delete actions.
4. ***Buttons with Bootstrap Icons:*** All actions such as edit, delete, and add command use Bootstrap icons to appear more professional and interactive.
5. ***Bot Display:*** The use of colored badges to indicate the active or inactive status of the bot.

This helps make the `manage_bots.html` page look more professional and organized, in line with styling preferences using Bootstrap without additional CSS.

Drawbacks of the Previous Code

The previous code used basic tables without additional designs that facilitate navigation and management. By using cards, responsive tables, and icons, the new design provides a better user experience. Additionally, the addition of badges for status and icons for platforms makes the information easier to understand and the tables more visually appealing. Action buttons with icons also enhance readability and accessibility.

15.3 Improving The Bot Edit Page

The Bot Edit Page is where users can update the details of existing bots. To ensure this page is effective and easy to use, we need to improve the layout of the form and the list of bot commands by providing better design elements and restructuring the page.

Design Enhancements for the Page

In the previous version, the Bot Edit page used basic forms and simple tables. To enhance the user experience, we can add several design elements, such as cards for a more structured appearance, icons to clarify functions, and rearranging elements to ensure ease of use.

Changes to the Code

1. ***Bot Edit Form:***
    - Use cards to wrap the form for a more structured and appealing layout.
    - Add icons and labels to each form group to provide clear context.
    - Use Bootstrap classes to make the form more attractive and responsive.
2. ***Form Layout:***
    - Organize form elements such as `bot_type`, `token`, `name`, `description`, `start_message`, and `help_message` using form groups and Bootstrap classes.

Complete Code after Refinement

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

{% block content %}
<div class="container mt-4">
    <div class="card shadow-sm border-0">
        <div class="card-header bg-primary text-
white">
            <h5 class="card-title">
                <i class="bi bi-pencil-square"></i>
Edit Bot: {{ bot.name }}
            </h5>
        </div>
        <div class="card-body bg-light">
            <form method="post">
                {% csrf_token %}

                <!-- Bot Name -->
                <div class="mb-3">
```

```
                    <label for="id_name" class="form-
label"><i class="bi bi-robot"></i> Bot Name</label>
                    {{ form.name }}
                </div>

                <!-- Bot Token (only for Telegram
bots) -->
                {% if bot.bot_type == 'telegram' %}
                <div class="mb-3">
                    <label for="id_token_telegram"
class="form-label"><i class="bi bi-key-fill"></i>
Telegram Token</label>
                    {{ form.token_telegram }}
                </div>
                {% elif bot.bot_type == 'whatsapp' %}
                <!-- WhatsApp Bot Token Fields would
go here -->
                {% endif %}

                <!-- Start Message -->
                <div class="mb-3">
                    <label for="id_start_message"
class="form-label"><i class="bi bi-play-circle"></i>
Start Message</label>
                    {{ form.start_message }}
                </div>

                <!-- Help Message -->
                <div class="mb-3">
                    <label for="id_help_message"
class="form-label"><i class="bi bi-info-circle"></i>
Help Message</label>
                    {{ form.help_message }}
                </div>

                <!-- Description -->
                <div class="mb-3">
                    <label for="id_description"
class="form-label"><i class="bi bi-file-earmark-
text"></i> Description</label>
                    {{ form.description }}
                </div>
                <!-- Submit and Back Buttons -->
                <div class="d-flex justify-content-
between mt-4">
                    <button type="submit" class="btn
btn-success">
                        <i class="bi bi-check-
circle"></i> Save Changes
                    </button>
                    <a href="{% url 'manage_bots' %}"
class="btn btn-secondary">
                        <i class="bi bi-arrow-left-
circle"></i> Back
                    </a>
                </div>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Explanation of Changes

1. ***Bot Edit Form:***
    - `card mb-4 shadow-sm` is used to wrap the form in a card with shadow for a cleaner look.
    - Added icons to form labels (`bi bi-tag`, `bi bi-key`, `bi bi-person`, `bi bi-info-circle`, `bi bi-chat-dots`, `bi bi-question-circle`) to enhance understanding.
    - Used radio button groups (`btn-group`) for the `bot_type` options with relevant icons, ensuring a more attractive and interactive display.
2. ***Form Layout:***
    - The form is organized with separate form groups for each element, using Bootstrap classes

(`form-control`, `border border-primary`) for better styling.

- `btn btn-success` for the submit button with a save icon (`bi bi-save`), enhancing the readability and functionality of the button.

Drawbacks of the Previous Code

The previous code used simple forms and tables without additional design elements. By using cards and improved styling, the layout of the Bot Edit page becomes more structured and appealing. The addition of icons and the use of Bootstrap classes make the form and command list easier to use and understand, enhancing the overall user experience.

15.4 Improving The Add Command Page

On this page, users can add new commands for the bot. To enhance the user experience and the aesthetics of the page, we need to make several design improvements. Here's how we can enhance the look and functionality of the Add Command page by using more modern and appealing design elements.

Design Enhancements for the Page

In the previous version, the Add Command page utilized a card with a basic design and simple form elements. To improve the appearance, we can add icons, use Bootstrap classes for better styling, and provide placeholders in the form elements to help users understand what to fill in.

Changes in the Code

1. ***Card Design:***
   - Using the `shadow-sm` class to give a subtle shadow to the card for better prominence.
   - Changing the card header color to blue with the `bg-primary` and `text-white` classes for better contrast.
2. ***Form:***
   - Adding icons to the form labels using `bi bi-terminal` for Command and `bi bi-chat-dots` for Response.
   - Using `border border-primary` classes to give borders to the input and textarea, enhancing the visibility of the form elements.
   - Adding placeholders in the input and textarea to provide users with hints about the data they need to fill in.
   - Changing the submit button to `btn btn-success` with the icon `bi bi-check-circle` for a more professional appearance.

Complete Code After Refinement

```
<!-- File: apps/templates/bots/add_command.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <!-- Header Section -->
    <div class="bg-primary text-white text-center p-4
rounded mb-4">
```

```
        <h1 class="display-6"><i class="bi bi-plus-
circle"></i> Add Command for Bot: {{ bot.name }}</h1>
    </div>

    <div class="card shadow-sm border-0">
        <div class="card-header bg-primary text-
white">
            <h5 class="card-title"><i class="bi bi-
terminal"></i> Add Command Form</h5>
        </div>
        <div class="card-body bg-light">
            <form method="post">
                {% csrf_token %}

                <!-- Form Group for Command Field -->
                <div class="mb-3">
                    <label
for="{{ form.command.id_for_label }}" class="form-
label"><i class="bi bi-code"></i> Command</label>
                    {{ form.command }} <!-- Default
Bootstrap styling for form input -->
                </div>

                <!-- Form Group for Response Field --
>
                <div class="mb-3">
                    <label
for="{{ form.response.id_for_label }}" class="form-
label"><i class="bi bi-chat-dots"></i>
Response</label>
                    {{ form.response }} <!-- Default
Bootstrap styling for textarea -->
                </div>

                <!-- Submit Button -->
                <div class="d-flex justify-content-
end">
                    <button type="submit" class="btn
btn-success">
                        <i class="bi bi-check-
circle"></i> Add Command
                    </button>
                    <a href="{% url 'manage_bots' %}"
class="btn btn-secondary ms-2">
                        <i class="bi bi-arrow-
left"></i> Back
                    </a>
                </div>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Explanation of Changes

1. ***Card Design:***
    - The `card shadow-sm` is used to provide a subtle shadow to the card, creating visual depth that makes the card appear more separated from the background.
    - `bg-primary` and `text-white` are used for the card header to make it more attractive and prominent, with clear contrast.
2. ***Form:***
    - Icons were added to the form labels (`bi bi-terminal` for Command and `bi bi-chat-dots` for Response) to provide clear visual context.
    - `form-control border border-primary` is used for the input and textarea to make them stand out and be clear with a blue border.
    - Placeholders were added to the input and textarea to provide visual hints about what users should fill in.

- The submit button was changed to `btn btn-success` with the icon `bi bi-check-circle`, adding green color and a check icon that indicates a successful action.

Drawbacks of the Previous Code

The previous code used a card with a basic design and form elements. Without icons, borders, and placeholders, the form's appearance did not provide sufficient guidance for users. With the addition of design elements such as icons, borders, placeholders, and button styling, this page is now more attractive and informative, making it easier for users to add new commands for the bot.

15.5 Improving The Edit Command Page

In this section, we will discuss improvements to the design of the Edit Command page to ensure that it is not only functional but also user-friendly and aesthetically pleasing. This page allows users to edit existing bot commands, and a good design will enhance the overall user experience.

Improvements and Explanations

We will enhance this page by implementing several design changes to improve aesthetics and functionality:

1. ***Use of Cards and Header:*** Utilize a card with a colored header to highlight the page title and improve visualization.

2. ***Adding Icons and Placeholders:*** Incorporate icons and placeholders in input fields and textareas to provide clearer guidance to users.
3. ***Form Styling:*** Use additional Bootstrap classes to enhance the styling of form elements.

Below is the revised code:

```
<!-- File: apps/templates/bots/edit_command.html -->

{% extends 'dashboard/base.html' %}

{% block content %}
<div class="container mt-4">
    <!-- Header Section -->
    <div class="bg-primary text-white text-center p-4
rounded mb-4">
        <h1 class="display-6"><i class="bi bi-pencil-
square"></i> Edit Command for Bot: {{ bot.name
}}</h1>
    </div>

    <div class="card shadow-sm border-0">
        <div class="card-header bg-primary text-
white">
            <h5 class="card-title"><i class="bi bi-
pencil"></i> Edit Command Form</h5>
        </div>
        <div class="card-body bg-light">
            <form method="post">
                {% csrf_token %}

                <!-- Form Group for Command Field -->
                <div class="mb-3">
                    <label
for="{{ form.command.id_for_label }}" class="form-
label"><i class="bi bi-code-slash"></i>
Command</label>
                    {{ form.command }} <!--
Displaying the input field without additional class
-->
```

```
                </div>

                <!-- Form Group for Response Field --
>
                <div class="mb-3">
                    <label
for="{{ form.response.id_for_label }}" class="form-
label"><i class="bi bi-chat-dots-fill"></i>
Response</label>
                    {{ form.response }} <!--
Displaying the textarea without additional class -->
                </div>

                <!-- Submit Button -->
                <div class="d-flex justify-content-
end">
                    <button type="submit" class="btn
btn-primary">
                        <i class="bi bi-check-circle-
fill"></i> Update Command
                    </button>
                    <a href="{% url 'manage_bots' %}"
class="btn btn-secondary ms-2">
                        <i class="bi bi-arrow-left-
circle"></i> Back
                    </a>
                </div>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Explanation of Changes:

1. ***Header and Card:***
   - ***`card shadow-sm`:*** Using a card with a shadow adds visual depth.
   - ***`card-header bg-primary text-white`:*** Provides an eye-catching header with a

blue background and white text, accompanied by an icon for contextualization.

2. ***Input Form:***
    - ***`form-control border border-primary`:*** Adds a colored border to input fields and textareas for better visibility.
    - ***Placeholder:*** Adds placeholders to provide instructions or examples of the expected data.
3. ***Submit Button:***
    - ***`btn btn-success`:*** Utilizes Bootstrap classes for a prominent green button and adds an icon to provide visual meaning to the button.

Drawbacks of the Previous Code:

1. ***Visual Design:*** The page appearance used basic Bootstrap elements but did not fully leverage Bootstrap's styling capabilities to enhance readability and aesthetics.
2. ***Icon Consistency:*** The code did not use icons to improve readability and provide better visual context.
3. ***Placeholders:*** There were no placeholders in input fields and textareas, which could confuse users regarding the expected data format.
4. ***Element Placement:*** Form elements and buttons could be refined to enhance the appearance and organization of the page.

These changes improve the user experience by providing a more engaging design and clearer functionality, ensuring that the Edit

Command page is easier to use and consistent with other pages in the application.

Chapter Conclusion

This chapter focuses on refining several key pages in the ***platform_bot*** application, particularly the pages related to bot management and their commands. By improving the design and functionality of these pages, we have enhanced the user experience and ensured that the application is not only functional but also enjoyable to use.

Key Points:

1. ***Navbar Page:***
    - ***Changes:*** Updated the navbar design to enhance readability and accessibility, while ensuring better responsiveness across various devices.
    - ***Objective:*** To provide clear and consistent navigation, which is essential for effective navigation within the dashboard.
2. ***Edit Bot Page:***
    - ***Changes:*** Implemented card design with improved styling, added icons to clarify functions, and refined form elements with borders and placeholders.
    - ***Objective:*** To provide a more professional and user-friendly interface for editing bot details, making it easier for users to input accurate data.
3. ***Add Command Page:***
    - ***Changes:*** Utilized card styling and a more prominent header, added icons and placeholders

for form elements, and refined buttons with consistent icons and colors.

- ***Objective:*** To simplify the process for users when adding new commands by providing a clear and intuitive interface, while enhancing the user experience with a more appealing design.

4. ***Edit Command Page:***
    - ***Changes:*** Added card design with a more prominent header, used icons and placeholders in input forms, and improved button styling.
    - ***Objective:*** To enhance the readability and usability of the bot command edit page, providing users with better control over managing existing commands.

The design enhancements on these pages focus not only on aesthetics but also on functionality and overall user experience. By applying consistent design principles, effectively using Bootstrap elements, and ensuring that the user interface is easy to navigate, the ***platform_bot*** application now offers a better and more professional experience. Improved design and clearer functionality assist users in managing bots and commands more efficiently, ultimately increasing productivity and user satisfaction.

Chapter 16 - Installing the Project on GitHub and Deploying Django

The process of modern web application development doesn't stop at just writing code. To ensure that the application is accessible online and properly managed, we need a reliable version control system and a robust hosting platform. One of the most popular solutions is to use GitHub as a version control platform and Django as a web application framework. After development is complete, the next step is to bring the application online so that users can access it through the internet. In this chapter, we will discuss in detail how to install a Django project on GitHub and then deploy it so that the application can be publicly accessible.

16.1 Introduction

Managing code without version control can become complicated, especially when working collaboratively or if the application continues to evolve with new features. This is why we need a version control system like Git and GitHub. Git is a tool to track changes in code, while GitHub provides a cloud-based platform to store and share code online. With GitHub, we can work in teams, manage code versions easily, and facilitate rollback in case of errors.

Using Git to store and manage our Django project offers many benefits. Not only can we track changes to the project, but we can also take advantage of GitHub as a place to share code with a team, collaborate, and showcase the project to the public.

Additionally, GitHub offers features like Issues, Pull Requests, and GitHub Actions for automation in the development pipeline.

On the other hand, deployment is an essential step that enables our application to be accessed by users via the internet. Without this process, the application we have developed cannot be directly used by users. Through deployment, we transform the Django application from a local environment to a server that can be accessed via a public domain. One popular method for deployment is using platforms like Heroku, PythonAnywhere, or a privately configured server.

In this chapter, we will dive deeper into how to use GitHub to manage our code and how to deploy it so that our Django application can be used online by users.

16.2 Uploading The Project To GitHub

After understanding the importance of version control and deployment, the first step we need to take is ensuring that our Django project code can be accessed and managed properly through GitHub. To do this, we need to upload the project to a GitHub repository. This repository will serve as an organized storage space for the code, allowing us to track changes and collaborate with other developers when necessary.

16.2.1 Creating a Repository on GitHub

The first step in this process is to create a new repository on GitHub. This repository will house all the files and folders of our project. Before uploading the Django project to GitHub, make sure that you already have a GitHub account and have set up Git on your local machine.

To get started, open the GitHub page and log into your account. Once logged in, navigate to the plus icon in the top-right corner of the page, then select ***New Repository***. Here, you will be prompted to provide a name for the repository. It's best to give the repository the same name as your Django project for easier recognition. For example, if your project is named *platform_bot*, you could name the repository *platform_bot* as well.

After deciding on a name, there are several important settings to consider:

1. ***Description*** – Add a short description of the project. This is optional, but it's helpful, especially if the repository is public.
2. ***Visibility (Public or Private)*** – Choose between making the repository public or private. If public, everyone can view your repository, which is useful for open-source projects. If private, only you and authorized collaborators can access the repository. For example, if this is a private or under-development project, you might choose the private option.

3. ***Initialize repository with a README*** – This option will automatically add a README file to the repository. A README is a text file that usually contains important information about the project, such as installation instructions or a brief explanation of its functionality.

After filling in all the information and selecting the settings, click the ***Create Repository*** button. Now, you have successfully created a new repository on GitHub. Next, we will connect this repository to our local Django project and upload the code to the repository.

16.2.2 Setting Up Personal Access Token (PAT) on GitHub

In this section, we will cover how to create and set up a Personal Access Token (PAT) on GitHub for authentication when interacting with Git repositories. Since GitHub no longer supports password authentication, we need to use a PAT as a replacement for a password in Git operations like push, pull, and clone via HTTPS.

A Personal Access Token is a type of digital key that represents your GitHub account permissions. With a PAT, you can access GitHub features such as repositories without having to enter your GitHub account password every time you perform a Git operation.

Creating a Personal Access Token on GitHub

The first step is to create a token for your GitHub account. Here's how to do it:

Log In to Your GitHub Account

First, log in to your GitHub account using a web browser. After logging in, access your account settings by clicking your profile picture in the top-right corner and selecting "Settings."

Open Developer Settings

In the settings page, scroll down the left-hand panel and find the "Developer settings" section. Click on it to proceed to the token access **settings**.

Create a Personal Access Token

- Under the "Developer settings" section, click on "Personal access tokens."
- Then, click the "Generate new token" button.
- You will be asked to provide a name or description for this token. Choose a clear name, such as "Token for Bot Platform Repository."
- Set the token’s expiration date. GitHub offers options for tokens to be valid for 30 days, 60 days, or longer.
  If you want the token to last indefinitely, you can select the no-expiration option, but be cautious about securing it properly.

Setting Token Permissions

When creating the token, you will need to choose the necessary access permissions (scope). To work with Git repositories, ensure that the `repo` option is selected, as this grants read and write access to your repository. If you also work with other features like packages, actions, or project management tools, you can select additional permissions as needed.

Generate the Token

After setting everything up, click on "Generate token." GitHub will then display the token that has been created. It’s important to copy this token immediately because, once the page is closed, the token cannot be viewed again. Store this token securely, for example, in a password manager.

Using a Personal Access Token

Once you have the PAT, the next step is to use it in your Git operations. Every time Git requests authentication (e.g., when running the `git push` command), you will enter this token in place of your GitHub password.

Example of Using a Personal Access Token in Git

For example, when trying to push to the repository, Git will prompt you for a username and password:

```
$ git push origin main
Username for 'https://github.com': username
Password for 'https://username@github.com': [Paste
your PAT here]
```

Instead of entering your GitHub account password in the password field, you will paste the token you created earlier. This will successfully authenticate you without requiring a password.

Storing the Personal Access Token Permanently

To avoid entering the token each time you run a Git operation, you can store the token locally on your computer using the Git Credential Manager. This way, Git will automatically use the token in the future.

Configuring Git to Save the Token

Run the following command in the terminal to configure Git to store authentication information:

```
$ git config --global credential.helper store
```

With this configuration, Git will save your token after you enter it for the first time, so you won't need to re-enter it every time you push, pull, or clone.

Using Git Credential Manager (Optional)

If it's not installed already, you can download and install the Git Credential Manager from the following link:
https://github.com/GitCredentialManager/git-credential-manager

Once the Credential Manager is installed, Git will automatically use the stored token.

In summary, we've learned how to create a Personal Access Token (PAT) on GitHub and use it as a password replacement for Git authentication. PATs provide better security and have become the new standard for accessing GitHub repositories via HTTPS. Always remember to store the token securely and revoke it when it's no longer needed.

16.2.3 Connecting a Local Repository to GitHub

After discussing the importance of version control and deployment in Django projects, the next step is to connect our local project to GitHub, so we can start storing and managing project changes online. To do this, we need to initialize Git in our project directory. This process aims to turn the project we're working on into a local Git repository that can later be synchronized with a repository on GitHub.

First, make sure that Git is installed on your computer. If it isn't installed yet, you can download and install Git from its official website. Once Git is installed, open a terminal or command prompt and navigate to your Django project directory. For example, if your project is in the `platform_bot` folder, you can run the following command to enter that folder:

```
$ cd platform_bot
```

Once inside the project directory, we need to initialize Git in this folder. The command used is as follows:

```
$ git init
```

This command will create a `.git` folder that serves as a storage for Git’s metadata in your project. The `.git` folder stores all the information needed to track changes to your code, such as commits, branches, and change history. After Git is initialized, you can start adding your project files to the staging area with the following command:

```
$ git add .
```

This command will add all the files in the project directory to the staging area, meaning that these files are ready to be committed. At this stage, Git hasn’t yet committed anything, but it has only marked the files as ready to be recorded. If you want to verify which files have been added, you can use the command:

```
$ git status
```

After adding the files, the next step is to make the first commit. A commit is a checkpoint in the repository that allows you to return to a specific version of the code if needed. To create a commit, you can run the following command:

```
$ git commit -m "Initial commit"
```

This commit will save a snapshot of your code at that moment, with the message "Initial commit" as a description of the changes. The commit message is very important, as it helps you and your team understand what has changed in each commit.

Once the commit is made, we need to connect this local repository to a GitHub repository. To do this, you must first create a new repository on GitHub. After the repository is created, GitHub will provide the URL for the repository. You need to add this URL to your local repository using the command:

```
$ git remote add origin
https://github.com/username/platform_bot.git
```

This command will link your local repository to the newly created GitHub repository. `origin` is the default name for a remote repository, but you can give it another name if necessary.

The final step is to push all your commits to the repository on GitHub. For this, you can use the command:

```
$ git push -u origin main
```

When running the `git push` command, Git will prompt you to enter your username and password.

Example:

```
$ git push origin main
Username for 'https://github.com': username
Password for 'https://username@github.com': [Paste
your PAT here]
```

In the **password** field, instead of entering your GitHub account password, you will paste the token you created earlier. This will successfully replace password authentication.

This command will send all commits from the main branch to the `origin` repository linked to GitHub. The `-u` option ensures that your local branch (`main`) will always be connected to the `main` branch on GitHub, so in the future, you only need to run `git push` to upload new changes.

Once this process is complete, your Django project will be successfully uploaded to GitHub, and any changes you make can be documented and managed through this version control system.

16.3 Setting Up A Server For Deployment

After successfully uploading our Django project to GitHub, the next step is to set up a server for deployment. Deployment is the process that allows our Django application to be accessible to the public via the internet. Choosing the right platform for deployment is important, as each platform has its advantages and disadvantages, depending on the needs and scale of the application we're building.

16.3.1 Choosing a Platform for Deployment

There are several platforms commonly used for deploying Django applications. These platforms generally offer cloud services that make it easier for us to host applications without having to worry about complex server configurations. Here are some common platforms used for Django deployment:

Heroku

Heroku is one of the most popular platforms for deploying web applications, including Django applications. The main advantage of Heroku is its ease of use. With just a few commands, our application can be deployed and accessed online. Heroku also offers good integration with GitHub, so every time we push to the GitHub repository, the app on Heroku can be automatically updated.

Heroku supports add-ons to enhance the application's functionality, such as PostgreSQL databases and email delivery services. However, Heroku's free version has limitations, such as slower response times and resource usage restrictions.

Virtual Private Server (VPS)

If we need full control over the server where the Django application is hosted, using a Virtual Private Server (VPS) is the right choice. A VPS gives us complete freedom to set up the server environment, from installing the operating system to configuring databases and web servers like Nginx or Apache.

A VPS offers more stable performance compared to platforms like Heroku because we get more dedicated resource allocation. However, using a VPS requires knowledge of server administration, including full responsibility for the security and performance of the server.

Railway

Railway is a relatively new cloud platform that is quickly gaining popularity among web developers. Railway offers an easy deployment process similar to Heroku. We can link our GitHub repository to Railway, and the platform will automatically deploy each time there are changes in the repository.

Railway provides flexibility in resource usage and comes with integrated database services. Its free version has limitations in terms of memory allocation and the number of projects that can be hosted.

PythonAnywhere

PythonAnywhere is a cloud platform specifically designed for Python applications, including Django. The advantage of PythonAnywhere is its very easy setup process, without the need for much manual server configuration.

We only need to upload the project code, set up the virtual environment, and configure webhooks and the database. This platform also provides HTTPS services, making it suitable for Django applications that require webhooks, such as Telegram and WhatsApp bot integration.

PythonAnywhere also offers fast and stable deployment options, with a free package that's sufficient for small projects. However, for larger-scale projects or if more control is needed, the paid plans may be more suitable.

16.3.2 Choosing a Platform Based on Needs

When choosing a platform for deployment, several factors need to be considered, such as the scale of the application, budget, and the level of control we need over the server. For a Django project that involves a bot with webhooks,

here are a few platform recommendations:

- ***Heroku***: The best option if you're looking for ease of deployment and don't want to deal with server configurations. Heroku supports webhook integration via HTTPS, making it ideal for bot projects like Telegram or WhatsApp.
- ***VPS (Virtual Private Server)***: If you need full control over the server, VPS services like DigitalOcean or AWS EC2 offer complete flexibility. VPS also allows for setting up Nginx or Apache servers to efficiently handle webhook requests.
- ***Railway***: Railway is suitable for developers who want a fast and simple deployment process, with good webhook support.
- ***PythonAnywhere***: PythonAnywhere offers ease of deployment for Django applications with minimal configuration. This platform is ideal for Python-based bot applications that require webhooks and HTTPS

services with simple setup. PythonAnywhere is a great choice if you're looking for a lightweight, easy-to-use platform specifically for Python projects.

16.3.3 Platform Recommendations

If you're looking for ease of deployment without needing complicated configurations, Heroku or PythonAnywhere could be the best choices, especially for small to medium-scale projects. However, if you need more control or if your bot application requires high performance, VPS will provide greater flexibility and scalability.

Choosing the right deployment platform is a crucial step to ensure your Django application is accessible to users in a stable and optimal manner.

16.4 Deploy To PythonAnywhere

To start the deployment process on PythonAnywhere, the first step is to create an account on the platform. PythonAnywhere offers a free option with some limitations that are suitable for developing and testing small projects like our Django project. If you don't have an account yet, you can visit the official PythonAnywhere website and follow the registration steps. Once registered, log in to the PythonAnywhere dashboard to continue.

16.4.1 Cloning the Repository from GitHub

After logging in, you'll be directed to the main dashboard. To begin setting up the deployment environment, open the bash terminal through New Console and select the Bash option. From this terminal, you can clone your Django project from GitHub to PythonAnywhere. For example, use the following command to copy the repository from GitHub:

```
$ git clone
https://github.com/username/platform_bot.git
```

Replace "username" with your GitHub username and "platform_bot" with the name of your project repository. This will copy all the files and project structure to PythonAnywhere.

16.4.2 Setting Up the Web App on PythonAnywhere

Once the repository is successfully cloned, you can proceed to configure the web application on PythonAnywhere. Go to the ***Web*** tab in the dashboard, then choose to add a new application. When prompted to select a framework, choose Django so that PythonAnywhere will automatically set up the Django environment. Make sure that PythonAnywhere detects the correct version of Python for your project.

16.4.3 Configuring Static Files and Templates

At this stage, it's important to configure the static files and template directories so that Django can serve files like CSS, JavaScript, images, and HTML templates correctly. This configuration is done in the ***Static files*** section of the Web tab. Add the following configuration for static files:

- ***URL***: `/static/`
- ***Directory***: `/home/username/platform_bot/apps/static/assets`

In addition to static files, you also need to ensure that the template directory is correctly set up. PythonAnywhere must know where the template files for your application are located. For templates, configure the directory like this:

- ***URL***: `/templates/`
- ***Directory***: `/home/username/platform_bot/apps/templates`

Replace "username" with your PythonAnywhere username. The `/static/` URL is used by the application to serve static files, and `/templates/` is used to access HTML template files. These settings ensure that both static files and templates are found and served correctly by the server.

16.4.4 Configuring the WSGI File

The next step is to ensure that PythonAnywhere knows where the WSGI file for your project is located. In the ***Source code*** section of the Web tab, input the path to your application's `wsgi.py` file. Typically, the path is:

```
/home/username/platform_bot/platform_bot/wsgi.py
```

This file is used by the server to run the Django application.

16.4.5 Reloading the Application

After completing all the configurations—static files, template directories, and WSGI path—the final step is to reload the application. In the ***Web*** tab, click the ***Reload*** button to apply all changes. By reloading, PythonAnywhere will read the new configuration and run the application with the latest setup.

Once reloaded, you can open the application’s domain provided by PythonAnywhere to verify if everything is running properly and if static files and templates are functioning as expected. If any issues arise, check the error logs, which can be accessed from the ***Log*** tab in the PythonAnywhere dashboard, to get further insights.

By following these steps, you have successfully deployed your Django application to PythonAnywhere and set up the static files and templates to be properly accessible.

16.5 Deploying To The Heroku Platform

16.5.1 Installing and Setting Up the Heroku CLI

After setting up the server for deployment, one of the most commonly used platforms for hosting Django applications is Heroku. Heroku offers ease in the deployment process and application management, making it an excellent choice for developers who want to focus on development without being burdened by complex server setups. To start, we need to install the Heroku CLI, which is a command-line tool that allows us to interact with the Heroku platform.

The first step is to visit the official Heroku website and create an account if we don't already have one. After registering, we can proceed to install the Heroku CLI. If you are using Windows, you can download the appropriate installer from the Heroku download page. For macOS or Linux users, you can use Homebrew or apt-get. For example, for macOS users, run the following command in the terminal:

```
$ brew tap heroku/brew && brew install heroku # Installing Heroku CLI with Homebrew
```

For Ubuntu users, you can use the following command:

```
$ curl https://cli-assets.heroku.com/install.sh | sh # Installing Heroku CLI on Ubuntu
```

After the installation is complete, we can verify that Heroku CLI is installed correctly by running the following command:

```
$ heroku --version # Check the Heroku CLI version
```

With the Heroku CLI installed, the next step is to log in to our Heroku account. Simply run the following command:

```
$ heroku login # Log in to Heroku
```

After running this command, a browser window will open, prompting us to enter our Heroku account credentials. Once successfully logged in, the terminal will confirm that we are logged in.

Next, we need to prepare our application for deployment. Ensure that we are in our Django project directory. Before deploying, we need to create a Procfile, which tells Heroku how to run our application. Inside the project directory, we can create this file with the command:

```
$ echo "web: gunicorn project_name.wsgi --log-file -"
> Procfile # Create Procfile for Heroku
```

Replace `project_name` with the name of our project. With this file, Heroku will know to run the Gunicorn server when our application is launched.

Next, we need to configure some settings in the `settings.py` file to ensure the application can run on Heroku. For example, we need to add Heroku to `ALLOWED_HOSTS`:

```
ALLOWED_HOSTS = ['.herokuapp.com']  # Add Heroku
domain
```

With all these preparations, we are ready to deploy our application to Heroku. This process will guide us in uploading and running the application online, allowing our bot to function correctly.

16.5.2 Setting Up Procfile and `requirements.txt`

After successfully logging in to Heroku and creating a new app, the next step is to set up two important files: the Procfile and `requirements.txt`. These two files play a crucial role in determining how our application will run on Heroku.

The Procfile is a text file that tells Heroku how to run our application. For a Django application using Gunicorn as the server, we need to ensure that the Procfile contains the correct command. To create this file, we can use the following command in the terminal, making sure we are in the project directory:

```
$ echo "web: gunicorn project_name.wsgi --log-file -"
> Procfile # Create Procfile for Gunicorn
```

Replace `project_name` with the name of our Django project. With this content, Heroku will run Gunicorn and load our application with the `wsgi.py` file.

Next, we need to ensure that all dependencies required to run our application are listed in the `requirements.txt` file. This file contains a list of Python packages necessary for our application to function properly. To generate the `requirements.txt` file, we can use the following `pip` command:

```
$ pip freeze > requirements.txt # Generate
requirements.txt
```

This command will record all the packages installed in our virtual environment into the `requirements.txt` file. After that, we need to ensure that this file includes Gunicorn, Django, and any other libraries we are using in the application. If we add new dependencies, we can update the file in the same way.

Now, with the Procfile and `requirements.txt` set up, we are ready to proceed with the deployment process. Heroku will use the information from these two files to correctly run our application on their servers. This is essential to ensure that our bot can function properly in a production environment.

16.5.3 Deploying to Heroku

Once all the important files, such as the Procfile and `requirements.txt`, are ready, we can proceed to the final step, which is connecting the Django project with Heroku and deploying it.

First, we need to create a new app on Heroku. To do this, make sure we are logged in to the Heroku account through the terminal. Then, we can create a new app using the following command:

```
$ heroku create # Create a new app on Heroku
```

After this command is executed, Heroku will automatically create a new app with a unique URL. This URL will be used to access our application online once the deployment is successful.

The next step is to connect the existing application in our GitHub repository to the newly created app on Heroku. To do this, we need to add Heroku as a remote to our local Git repository. Use the following command inside our project directory:

```
$ git remote add heroku https://git.heroku.com/your-app-name.git # Connect Heroku to local repository
```

Replace `your-app-name` with the name of the app generated by Heroku. Now, our application is connected to Heroku, and we can easily deploy it.

To push the code from the `main` branch to Heroku and start the deployment process, we can run the following command:

```
$ git push heroku main # Deploy the application to Heroku
```

This process will upload all our project files to Heroku and start building the application. Heroku will read the Procfile to understand how to run the app and the `requirements.txt` to install all necessary dependencies. If there are no errors, the app will soon run on Heroku's servers, and we can access it via the previously provided URL.

With this command, our Django application has been successfully deployed to Heroku and is ready to be accessed online. The bot we built can now start operating, receiving and responding to requests via the webhook we set up earlier. Heroku provides a simple solution for deployment with efficient steps, making this platform an ideal choice for running Django-based bots.

16.5.4 Handling Environment Variables

At this point, after successfully deploying our Django application to Heroku, there is one crucial step left to ensure the application runs securely and with the correct configuration. We need to set up the environment variables used by Django, such as `SECRET_KEY`, `DEBUG`, and `DATABASE_URL`. Managing these variables directly in the code is not recommended because it can pose security risks, especially with the `SECRET_KEY`.

Heroku provides an easy way to handle environment variables through a feature called "Config Vars." With Config Vars, we can store sensitive values like `SECRET_KEY` without needing to keep them in our code. To set up environment variables in Heroku, we can do so with a few steps.

First, ensure we are logged in to Heroku through the terminal, then add environment variables like `SECRET_KEY` using the following command:

```
$ heroku config:set SECRET_KEY=super-secret-key # Set SECRET_KEY
```

This command will add the `SECRET_KEY` variable to the Heroku app. Replace `super-secret-key` with a truly secret and secure key. In addition to `SECRET_KEY`, we also need to set other important variables such as `DEBUG` and

`ALLOWED_HOSTS`. An example command to set the `DEBUG` variable so that our app runs in production mode is:

```
$ heroku config:set DEBUG=False # Set production mode
```

For the `ALLOWED_HOSTS` variable, we can add the Heroku domain created when we ran the `heroku create` command.

For example, if our app's URL is `my-app.herokuapp.com`, we can set it like this:

```
$ heroku config:set ALLOWED_HOSTS=my-
app.herokuapp.com # Set ALLOWED_HOSTS
```

Additionally, if our Django app uses a webhook to operate the bot, we need to set the variable related to the webhook URL. For instance, if we have defined `WEBHOOK_URL` in `settings.py`, we can add it to Heroku with the command:

```
$ heroku config:set WEBHOOK_URL=https://my-
app.herokuapp.com/webhook/ # Set webhook URL
```

Once these environment variables are set, Heroku will automatically use them when running the application. We no longer need to worry about security issues since these sensitive variables will not appear in the code we commit to GitHub.

Keep in mind that every time there is a change to the environment variables, Heroku will use the updated values without requiring a manual restart. This makes the deployment process more efficient and secure, ensuring that important keys and configurations do not leak into public repositories.

With this setup, our Django application is ready to operate with the correct configuration, ensuring both security and performance.

16.6 Obtaining The Webhook URL

Webhook, as discussed in previous chapters, is a mechanism that allows an application to receive real-time notifications from an external server when certain events occur. In the context of bots, a webhook is useful for receiving updates or commands directly from the platform sending them, such as Telegram or other services. This enables the bot to respond to commands or notifications quickly without having to continuously poll the server.

Essentially, a webhook "connects" our bot application to an external server, and whenever relevant activity occurs, the server sends data to the webhook URL we specify. Therefore, the webhook URL is crucial because it acts as the gateway for communication between our bot and the server.

16.6.1 How to Obtain a Webhook on Heroku

Once our bot application is deployed to Heroku, the platform automatically provides a URL that can be used as a webhook. The URL has the format https://app-name.herokuapp.com/, which becomes the webhook address that we register with bot platforms like Telegram.

To obtain the webhook URL on Heroku, the first step is to ensure the application is deployed and accessible online. Once the app is running, we can use the URL provided by Heroku as the webhook.

For example, if our Heroku app is named "my-bot-app," the webhook URL will be:

```
https://my-bot-app.herokuapp.com/webhook/
```

This URL will then be registered with the bot platform, such as Telegram. For Telegram, we need to send an API request to the Telegram endpoint with the following format:

```
$ curl -F
"url=https://my-bot-app.herokuapp.com/webhook/"
https://api.telegram.org/bot<TOKEN>/setWebhook
```

Make sure to replace `<TOKEN>` with the bot API token obtained when creating the bot in Telegram. This command links the webhook to the bot, so any messages or commands sent to the bot will be forwarded to our application through the specified webhook URL.

Besides Telegram, this process can be applied to other platforms with the appropriate webhook format and endpoint. Using a webhook ensures our bot application is always ready to receive real-time notifications from the service in use.

After the webhook is set up and connected to the Heroku app, we can verify that the webhook is functioning correctly by sending a test message to the bot and checking if our application responds appropriately.

16.6.2 Activating Webhook for the Bot

After obtaining the webhook URL from Heroku, the next step is to link this webhook to the bot we’ve created. This process involves registering the webhook URL with the bot server being used, such as Telegram.

By activating the webhook, the bot will receive messages or notifications from the server whenever there is related activity, such as a user sending a message to the bot.

For a Telegram bot, we can use Telegram’s API to quickly register the webhook. Telegram provides an endpoint to do this, and we can send an API request to link the webhook URL to our bot.

The first step is to ensure we have the webhook URL from Heroku, which has the format:

```
https://<app_name>.herokuapp.com/<bot_endpoint>/
```

Where `<app_name>` is the name of our application on Heroku, and `<bot_endpoint>` is the route within Django we have set to receive notifications from the webhook.

To activate the webhook, we can use the `curl` command to send a request to the Telegram API. Make sure to replace the bot token and URL with our configuration:

```
$ curl -F
"url=https://my-bot-app.herokuapp.com/webhook/"
https://api.telegram.org/bot<token>/setWebhook
```

In this command, `<token>` is the unique token obtained when creating the bot on Telegram, and `https://my-bot-app.herokuapp.com/webhook/` is the URL that will receive updates from Telegram.

After running this command, Telegram will forward the messages received by the bot to this URL. We can check whether the webhook is correctly connected by inspecting the response from the Telegram API. If the webhook is successfully activated, the Telegram API will return a success status.

To verify that the webhook is working, we can send a test message to the bot and see if the application receives and processes it as expected. If everything works, our bot is ready to receive real-time messages through the webhook without needing continuous polling.

With this step, our bot is fully connected to the webhook and ready for online use, responding to every interaction sent to the registered webhook URL.

Chapter Conclusion

In this chapter, we have learned the essential steps to manage and distribute a Django project online, starting from uploading the project to GitHub to deploying it on platforms like Heroku. The first step we took was linking the local project with a repository on GitHub, which provides great advantages in terms of version control and collaboration.

Next, we explored the deployment process, including choosing a platform like Heroku, setting up the deployment environment, and configuring the `Procfile` and `requirements.txt` to ensure the application runs smoothly on the production server. We also emphasized the importance of setting environment variables, such as `SECRET_KEY`, on the deployment platform to maintain the application's security.

Equally important, we discussed how to obtain and activate a webhook, which is a crucial component for bot operations. The webhook ensures that the application can receive and process messages in real-time from users without the need for intensive polling. By linking the webhook URL to the bot via Telegram's API, we have configured the bot to run efficiently in a production environment.

This entire chapter provides a comprehensive guide on how a Django application, particularly a bot application, can be set up and run online. The deployment process opens the door for our application to be accessed globally, expanding its reach and ensuring the bot's functionality operates optimally on a stable server. Ultimately, we are ready to launch the Django project into the real world using the available tools and effective techniques to ensure everything runs smoothly.

Conclusion

In this book, we have explored various aspects of developing Django applications, particularly in the context of building bot management platforms. From understanding the fundamentals of Django to implementing more complex features such as token validation and integration with various bot platforms, each chapter has been designed to provide readers with comprehensive and practical insights into Django-based application development.

Throughout this journey, we also covered how to improve the appearance of the dashboard, set up an activity log system, and deploy projects to platforms like GitHub and Heroku. We learned how to optimize the user interface using various Bootstrap components, and how to structure a well-organized project architecture to support sustainable development. With this detailed guide, it is hoped that readers, whether they are new to Django or more experienced, can develop their own projects more systematically and efficiently.

Additionally, this book emphasizes the importance of staying up to date with the latest developments in the Django ecosystem and web technologies in general. By leveraging official documentation, tutorials, and community forums, readers are encouraged to continue learning and exploring new possibilities in application development. In a constantly changing world, the ability to adapt and develop new skills is key to success.

Thus, this book is not just a technical guide, but also aims to inspire anyone who wishes to further explore the world of web development using Django. I hope that each chapter discussed helps spark new ideas and encourages readers to create innovative and useful applications. I hope this book serves as a solid foundation for your journey towards more advanced and impactful application development.

Hopes For The Reader

I hope this book provides valuable insight and knowledge for readers in developing Django-based applications, particularly in the context of bot management. By understanding the concepts and practices explained, I hope readers can:

1. ***Find Inspiration***: Use the information and techniques provided to develop new and innovative ideas in their own projects. The world of web development offers many opportunities, and I hope readers find the passion to keep exploring and experimenting.
2. ***Improve Skills***: Each chapter is designed not only to impart theoretical knowledge but also to practice skills in application development. I hope readers can apply what they've learned in the real world and deepen their technical expertise.
3. ***Engage with the Community***: I encourage readers to get involved with the Django community and other developers. Sharing experiences, asking questions, and

collaborating can open the door to deeper learning and expand their professional networks.

4. ***Face Challenges***: In the process of developing applications, challenges are inevitable. I hope readers view challenges as opportunities to learn and grow, and that they can face each obstacle with confidence.
5. ***Innovate***: Finally, I hope readers become innovators in their fields. With the knowledge and skills gained from this book, I believe readers can create applications that are beneficial and have a positive impact on their communities.

With these hopes, I would like to remind you that the journey in software development is a continuous process. May this book be the first step for readers to keep learning and growing in this dynamic world of technology.

Final Project Structure

This Django project is named ***platform_bot***, designed to build and manage a bot management platform. The project structure is divided into several important parts, each with specific functionalities.

Here is the built Django project structure, which is expected to help readers understand how each component interacts with one another:

```
platform_bot/
│
├── apps/
```

```
│   ├── context_processors.py       # Contains
functions to add additional context to templates.
│   ├── authentication/              # Application
for managing user authentication processes (login,
register).
│   ├── dashboard/                   # Application
that manages the dashboard view for users.
│   ├── users/                       # Application
for managing user profiles.
│   ├── webhook/                     # Application
for handling webhooks from bots.
│   ├── bots/                        # Application
that manages all aspects related to bots.
│   │   ├── services/                # Contains
services to handle integration with bot platforms.
│   │   │   ├── telegram.py          # Service for
interaction with the Telegram API.
│   │   │   ├── whatsapp.py          # Service for
interaction with the WhatsApp API.
│   ├── static/                      # Stores static
files such as CSS, JavaScript, and images.
│   │   ├── assets/                  # Folder for
storing static assets.
│   │   │   ├── css/                 # Contains CSS
files for styling.
│   │   │   │   ├── style.css        # Main CSS file
for the application.
│   │   │   ├── js/                  # Contains
JavaScript files for interactivity.
│   │   │   │   ├── script.js        # Main
JavaScript file for the application.
│   └── templates/                   # Stores HTML
template files for rendering.
│       ├── home/                    # Folder for the
home page.
│       │   ├── base.html            # Base template
for the home page.
│       │   ├── home.html            # Main home page
of the application.
│       ├── account/                 # Folder for
user account-related pages.
│       │   ├── register.html        # Page for
registering new users.
│       │   ├── login.html           # Page for user
login.
│       ├── dashboard/                # Folder for
dashboard pages.
```

```
│       │   ├── base.html           # Base template
for the dashboard.
│       │   ├── sidebar.html         # Template for
the sidebar in the dashboard.
│       │   ├── navbar.html          # Template for
the navbar in the dashboard.
│       │   ├── dashboard.html       # Main
dashboard page.
│       ├── users/                  # Folder for
user profile pages.
│       │   ├── profile.html        # Page to
display and edit user profiles.
│       ├── bots/                   # Folder for
bot-related pages.
│       │   ├── create_bot.html     # Page for
creating a new bot.
│       │   ├── manage_bots.html     # Page for
managing the bot list.
│       │   ├── edit_bot.html       # Page for
editing existing bot information.
│       │   ├── add_command.html    # Page for
adding new commands to the bot.
│       │   ├── edit_command.html   # Page for
editing existing commands.
├── platform_bot/
│   ├── __init__.py                 # Marks this
folder as a Python package.
│   ├── asgi.py                      # Configuration
for ASGI, needed for asynchronous support.
│   ├── settings.py                  # Contains
settings and configurations for the Django project.
│   ├── urls.py                      # Stores URL
settings for the project.
│   └── wsgi.py                      # Configuration
for WSGI, for running the application.
│
├── manage.py                        # Script to
manage the Django project (run server, migrations,
etc.).
└── db.sqlite3                      # SQLite
database for storing application data.
```

Explanation of Each Part

- ***apps/***: This folder contains separate applications that handle various aspects of the project. Each application has a different functional focus, such as user authentication, dashboard management, and bot integration.
- ***static/***: Stores static files used by the application, including CSS and JavaScript files that enhance the application's appearance and interactivity.
- ***templates/***: Contains HTML files used for rendering the application's views. Templates are divided into several categories based on functionality, such as home pages, authentication, dashboards, and bot management.
- ***platform_bot/***: This is the main project folder that stores the core settings and configurations of the Django application, including settings files, URLs, and server management.
- ***manage.py***: This script is used to run project management commands, such as starting the server, performing database migrations, and more.
- ***db.sqlite3***: The SQLite database that stores all application data, including user information, bots, and added commands.

By understanding this project structure, readers are expected to see how the various components work together to create a functional and efficient bot management platform. Each part of the project plays an important role and integrates with one another, enabling more structured and maintainable application development.

GitHub Project Link

Additionally, you can view the bot management platform project discussed in this book at the following GitHub link:

GitHub Project - Platform Bot

Through this repository, you can access the complete source code of the project we developed in this book, as well as contribute if you're interested in further developing it.

ACKNOWLEDGEMENTS

I would like to express my gratitude to all parties who have supported me throughout the writing process of this book. Thank you to my family, friends, and fellow developers who have always provided motivation and inspiration. Their moral support and presence during this journey have meant a lot to me, and I would not have been able to complete this book without the encouragement they provided.

I would also like to extend a special thank you to the Django community, which is always active in sharing knowledge and valuable resources. This community is not only a place to ask questions and learn but also a source of invaluable ideas and innovations. Without a strong community, the development of open-source applications like Django would not be as extensive and impactful as it is today. The existence of forums, documentation, and various tutorials from community members has greatly helped me deepen my understanding and skills.

I am also grateful to all the readers who have taken the time to read this book. Your engagement and feedback mean a lot to me. I hope this book can provide significant benefits for your journey in Django application development. May every concept discussed, from the basics to advanced features, serve as useful tools in achieving your development goals.

In closing, I encourage readers to continue exploring and learning. The world of information technology is constantly evolving, and being a lifelong learner is one of the keys to staying relevant. Do not hesitate to share the knowledge you have gained with others, as collaboration and sharing information are effective ways to strengthen our community. May we all contribute together to a better and more innovative web development world. Thank you.

Thank you for following this journey, and I wish you success in all your future projects!

dccli

BUILDING A BOT PLATFORM WITH DJANGO

STEP-BY-STEP GUIDE TO BUILDING A BOT BUILDING PLATFORM FROM SCRATCH

THIS BOOK GOES INTO DEPTH ON HOW TO BUILD A BOT PLATFORM USING THE DJANGO FRAMEWORK. WITH A PRACTICAL APPROACH AND REAL-WORLD EXAMPLES, READERS WILL BE INVITED TO LEARN ABOUT THE DEVELOPMENT OF BOTS THAT CAN COMMUNICATE THROUGH PLATFORMS SUCH AS TELEGRAM AND WHATSAPP, USING WEBHOOKS AS AN EFFECTIVE METHOD OF INTERACTION. FROM DESIGN, IMPLEMENTATION, TO DEPLOYMENT TO SERVERS, THIS BOOK IS THE PERFECT GUIDE FOR ANYONE LOOKING TO GET STARTED OR DEEPEN THEIR BRAND UNDERSTANDING

https://aigoretech.my.id