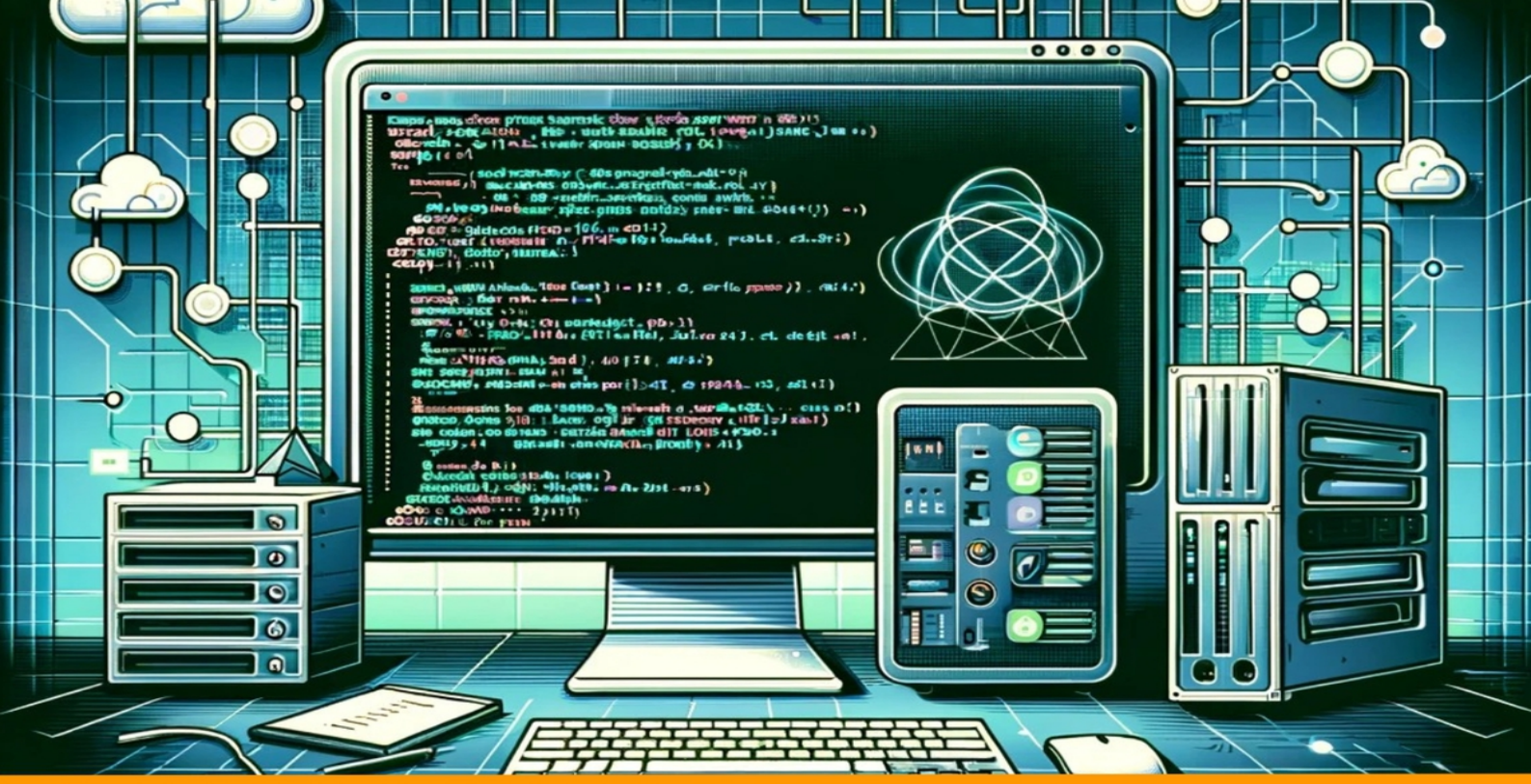

BUILD A BACKEND REST API WITH PYTHON & DJANGO - BEGINNER

Create that App & fully-functioning user database in this crash course to building a REST API.

Contents Guide

Section 1:

Introduction

Course Overview

Welcome to the introductory chapter of “Build a Backend REST API with Python & Django - Beginner”. This chapter is designed to provide you with an overview of what to expect as you journey through this course. By the end, you should have a clear idea of what topics will be covered, and you’ll be prepared to dive into the more hands-on sections that follow.

1.1 Purpose of the Course

The primary purpose of this course is to guide you in building a robust, secure, and scalable backend REST API using the Django and Django REST Framework (DRF). The knowledge and skills you acquire here are essential in today’s world of app and web development. Whether you’re a startup wanting to launch the next big thing or a developer looking to expand your skill set, mastering the art of creating a backend can

exponentially increase the potential and power of your applications.

1.2 Why Django and Django REST Framework?

Django is a high-level, open-source web framework written in Python. It follows the “Don’t repeat yourself” (DRY) principle and emphasizes the reusability of code. This makes it one of the most efficient frameworks to develop web applications.

The Django REST Framework, or DRF, is a powerful toolkit built on top of Django, specifically designed to build Web APIs. It provides you with the tools to make data serialization, authentication, view sets, routers, and more a breeze. The combination of Django and DRF is a proven stack for developing modern web APIs.

1.3 Course Highlights

Throughout the course, we’ll be covering the following:

- Setting Up Development Environment: Before we dive into coding, we’ll ensure that your development environment, regardless of whether you’re on Windows or macOS, is ready and equipped with all the necessary tools.

- Understanding and Implementing Django: From setting up your very first Django project to creating complex data models and integrating with Django’s admin site, you’ll learn the ins and outs of this amazing framework.

- Diving Deep into Django REST Framework: Building on your Django knowledge, you’ll be introduced to DRF’s advanced capabilities, from creating APIViews to implementing full-fledged Viewsets.

- Building an End-to-End REST API: This isn’t just a theoretical course. By the end, you’ll have built a complete REST API capable of handling user profiles, authentication, status updates, and more.

- Deploying to AWS: Once your API is ready, we'll walk you through the process of deploying it on AWS, one of the world's leading cloud service providers.

1.4 Who Should Take This Course

This course is tailored for:

- Developers aiming to build a backend for their apps or MVPs.

- Frontend developers eager to learn backend to become full-stack.

- Beginners in the tech field looking to enhance their portfolio and career prospects.

1.5 Prerequisites

While the course is designed with beginners in mind, having a basic understanding of Python can be advantageous. However, don't worry if you're completely new. The course is structured to take you from the basics upwards, ensuring everyone can keep pace.

1.6 Conclusion

This overview provides a sneak peek into the exciting journey you're about to embark on. With dedication and focus, by the end of this course, you will not only have a functioning REST API but also a profound understanding of backend development, giving you a competitive edge in the tech industry.

Now, let's roll up our sleeves and delve into the world of backend development! In the next chapter, we'll discuss two popular development tools: Vagrant and Docker, helping you make an informed decision on which to use.

Vagrant vs Docker

Objective: By the end of this chapter, readers will have a clear understanding of the differences between Vagrant and Docker, their use cases, and which might be more suitable for specific scenarios in web development.

Assets, Resources, and Materials:

1. Vagrant:

   - Acquisition: Download from [Vagrant's official website](https://www.vagrantup.com/)

   - Use: Virtualization tool to create and manage virtualized development environments.

2. Docker:

   - Acquisition: Download from [Docker's official website](https://www.docker.com/)

   - Use: Platform to develop, ship, and run applications inside containers.

Introduction

In the modern development world, ensuring that the software you build runs consistently across all environments is crucial. Both Vagrant and Docker aim to solve this issue but in slightly different manners. To understand which is better suited for your needs, it's essential to know the basics of each and how they differ.

What is Vagrant?

Vagrant is a tool that allows developers to create, configure, and manage a complete virtualized environment. At its core, Vagrant uses virtualization technologies like VirtualBox, VMware, and others to provide an environment that matches production as closely as possible.

Key Features of Vagrant:

- Reproducible Environments: By utilizing Vagrantfiles, a scripted definition of the virtual machine, Vagrant

ensures the environment remains consistent.

- Provider Flexibility: While it integrates best with VirtualBox, Vagrant supports various providers like AWS, VMware, and more.

- Networking: Easily simulate complex network environments within your virtual machine.

What is Docker?

Docker, on the other hand, is all about containers. A container is a standalone package of software that includes everything required to run it: code, runtime, system tools, libraries, etc. Containers are isolated but run on a shared OS, making them lightweight compared to virtual machines.

Key Features of Docker:

- Lightweight: Containers share the host system OS kernel, removing the need for an OS per application.

- Docker Hub: A cloud-based registry service that allows you to link code repositories, build details, and more.

- Portable: Docker containers can run anywhere - on a developer's machine, on physical hardware, virtual machines, public clouds, private clouds, and more.

- Microservices: Docker's architecture is inherently designed to support microservices, a modern software design pattern.

Comparing Vagrant and Docker

1. Overhead and Performance: Docker containers are lightweight since they share the host OS. In contrast, Vagrant creates a full virtual machine, which can consume more resources.

2. Isolation: Vagrant provides better isolation because it runs separate OS instances for each environment. Docker containers, while isolated, share the same OS kernel.

3. Ecosystem and Community: Docker has a broader community and ecosystem, especially with the popularity of container orchestration tools like Kubernetes. Vagrant's ecosystem is more focused on VM providers and provisioning tools.

4. Portability: Docker's "build once, run anywhere" approach ensures that your application will run the same regardless of where the Docker container is launched. Vagrant environments, while reproducible, might still face minor differences depending on the provider.

5. Learning Curve: Docker can have a steeper learning curve, especially when diving into advanced features. Vagrant is simpler and more straightforward for setting up reproducible development environments.

When to Use Which?

- Vagrant: If you require full virtual machines or need to simulate intricate network configurations, or if your application necessitates different operating systems, Vagrant might be a better choice.

- Docker: If you're building microservices, need a more extensive ecosystem for scalability and orchestration, or simply want a lightweight and fast environment, Docker is the way to go.

Conclusion

Both Vagrant and Docker are powerful tools designed to make developers' lives easier. The choice between them should be dictated by the specific requirements of your project. While Vagrant is excellent for setting up virtual machines that mirror production servers, Docker's lightweight containers are changing the way we think about deployment and scalability.

As you dive deeper into backend development, you'll find scenarios where one might be preferable over the other.

Remember, the best tool is always the one that makes your job easier and your application more robust.

Section 2:

Setting up your Development Environment

Windows: Installing Git, VirtualBox, Vagrant, Atom and ModHeader

Assets/Resources Required for This Chapter:

- A stable internet connection.

- A Windows PC.

- Administrative privileges on your PC.

---

1. Git *(Source Control Management)*

How to Acquire: [Git Official Website](https://git-scm.com/)

Purpose: Git is a free and open-source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Steps to Install Git:

1. Visit the official website and click on the “Download” button for Windows.

2. Once the download is finished, run the installer.

3. Choose a location for installation. The default is usually fine.

4. Select the components you want to install. For most users, the default components are adequate.

5. Choose the default editor for Git. For beginners, the Windows default should be okay.

6. Adjust your PATH environment. I recommend “Use Git from the Windows Command Prompt” for easy access.

7. Choose HTTPS transport backend.

8. Check out Windows-style, commit Unix-style line endings option.

9. Use MinTTY as the default terminal emulator.

10. Finish the installation.

---

2. VirtualBox *(Virtualization Tool)*

How to Acquire: [VirtualBox Official Website](https://www.virtualbox.org/)

Purpose: VirtualBox is a powerful virtualization product for enterprise and home use.

Steps to Install VirtualBox:

1. Navigate to the VirtualBox official website and download the Windows installer.

2. Run the installer.

3. Confirm the default settings throughout the installation.

4. Complete the installation.

---

3. Vagrant *(Development Environment Manager)*

How to Acquire: [Vagrant Official Website](https://www.vagrantup.com/)

Purpose: Vagrant is a tool for building and managing virtual machine environments in a single workflow.

Steps to Install Vagrant:

1. Visit the official Vagrant website and download the Windows version.

2. Double-click on the installer.

3. Follow the installation wizard. Keep the default settings.

4. Finish the installation. Restart may be necessary.

---

4. Atom *(Text Editor)*

How to Acquire: [Atom Official Website](https://atom.io/)

Purpose: Atom is a free and open-source text and source code editor with support for plug-ins written in Node.js.

Steps to Install Atom:

1. Navigate to Atom's official website.

2. Download the Windows installer.

3. Once downloaded, run the installer. The installation should start automatically.

4. Once installed, Atom should automatically open. If not, you can find it in your applications folder.

---

5. ModHeader *(Browser Extension for Modifying HTTP Headers)*

How to Acquire: ModHeader is available on the Chrome Web Store.

Purpose: ModHeader allows you to modify request and response headers when testing web applications.

Steps to Install ModHeader:

1. Open Google Chrome and navigate to the Chrome Web Store.

2. Search for "ModHeader".

3. Click “Add to Chrome” on the ModHeader extension.

4. Confirm any prompts.

Conclusion:

Setting up a development environment requires careful attention to detail. Following the steps outlined above will ensure that you have the necessary tools installed to proceed with the next chapters of this course. Each of these tools will play a crucial role as we delve into creating a backend REST API using Django. Ensure everything is correctly installed before proceeding.

macOS: Installing Git, VirtualBox, Vagrant, Atom and ModHeader

Assets, Resources, and Materials:

- macOS computer: This is essential as the chapter’s installation instructions are tailored to macOS.

- Git: A distributed version control system used by developers to track changes in source code.

- VirtualBox: A free and open-source virtualization software package.

- Vagrant: A tool for building and distributing development environments.

- Atom: A free and open-source text editor that’s modern, approachable, and customizable.

- ModHeader: A browser extension used for modifying HTTP request and response headers.

(How to acquire: Can be added to browsers from the browser’s extension or plugin store. For this chapter, we will focus on adding it to Chrome. Available on [Chrome

Web Store](https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkpgkljpfnnfcklj?hl=en))

Introduction:

Setting up a development environment is the first essential step before delving into backend development. In this chapter, we'll guide you through the installation of crucial tools and software, ensuring you have everything you need to build a robust backend system.

---

1. Installing Git:

Purpose: Git allows developers to track and manage changes to their projects, facilitating collaboration and version control.

Steps:

1. Visit [Git's official website](https://git-scm.com/download/mac) and download the latest version for macOS.

2. Once downloaded, double-click the `.dmg` file to start the installation.

3. Drag the Git icon to your Applications folder.

4. To confirm successful installation, open Terminal and type:

"`bash

git —version

"`

You should see the installed Git version displayed.

---

2. Installing VirtualBox:

Purpose: VirtualBox lets you run multiple virtual machines on your macOS, which is essential for creating isolated development environments.

Steps:

1. Go to [VirtualBox's official website](https://www.virtualbox.org/wiki/Downloads) and select "OS X hosts" to download the installer.

2. Open the downloaded `.dmg` file.

3. Double-click the VirtualBox.pkg icon to start the installation and follow the on-screen instructions.

4. Once installed, you can access VirtualBox from your Applications folder.

---

3. Installing Vagrant:

Purpose: Vagrant works with VirtualBox to automate the setup of virtual environments, streamlining the development process.

Steps:

1. Navigate to [Vagrant's official site](https://www.vagrantup.com/downloads.html) and download the macOS version.

2. Open the `.dmg` file and drag the Vagrant icon to your Applications folder.

3. To verify the installation, open Terminal and type:

"`bash

vagrant —version

"`

The version of Vagrant you installed should be displayed.

---

4. Installing Atom:

Purpose: Atom is a versatile text editor optimized for coding, offering syntax highlighting, auto-completion, and integration with Git.

Steps:

1. Visit [Atom's official website](https://atom.io/) to download the macOS version.

2. Open the downloaded `.zip` file which will extract the Atom application.

3. Move the Atom application to your Applications folder.

4. Launch Atom to confirm successful installation.

---

5. Adding ModHeader Extension to Chrome:

Purpose: ModHeader allows you to modify and add HTTP headers, vital for testing and debugging APIs.

Steps:

1. Open Chrome and navigate to the [ModHeader extension on Chrome Web Store](https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkpgkljpfnnfcklj?hl=en).

2. Click "Add to Chrome" and confirm the addition.

3. Once added, you will see the ModHeader icon at the top-right corner of your Chrome browser.

Conclusion:

Congratulations! You have successfully set up essential tools for your development environment on macOS. These tools will be pivotal as you delve deeper into backend development in the subsequent chapters. Ensure you familiarize yourself with each software as they will be integral in the development and testing of your REST API.

Section 3:

Setting up your Project

Creating a workspace

Assets, Resources, and Materials:

1. Computer: Any modern computer (Mac, Windows, or Linux) will suffice.

2. Operating System: Ensure you have an up-to-date version of Windows, macOS, or Linux.

3. File Explorer: Built-in explorer in your OS (e.g. Windows Explorer for Windows, Finder for macOS).

4. Text Editor: Atom (You can download it from atom.io). We will use Atom to write and manage our code.

5. Dedicated Folder: A clean directory/folder where all your project files will reside.

Introduction:

Before diving into the actual coding and server configurations, it's paramount to set up a clean and organized workspace on your computer. A dedicated workspace will make it easier to manage files, track changes, and collaborate with others if needed. In this chapter, we will guide you through the process of creating an organized workspace for your Django REST API project.

Steps to Create a Workspace:

1. Choose a Suitable Location on Your Computer:

Begin by deciding where you want to store your project. It could be in your user directory, a dedicated `projects` directory, or any other location you find convenient. Just ensure you have enough space and easy access to it.

2. Create a New Directory for Your Project:

Navigate to the chosen location using your File Explorer. Create a new directory (or folder) and name it descriptively, like `django_rest_api_project`. This name will help you identify the project at a glance in the future.

3. Open Atom Text Editor:

Launch the Atom text editor. If you haven't already downloaded Atom, you can get it from atom.io.

4. Add Your Project Directory to Atom:

- In Atom, go to `File` > `Add Project Folder`.

- Navigate to the `django_rest_api_project` directory you just created and select it.

- You should now see the directory in Atom's sidebar, confirming that it's been added as a project folder.

5. Creating Essential Sub-Directories:

For the sake of organization, create a few sub-directories within your main project folder:

- `source`: This will hold all the source code for your project.

- `docs`: For any documentation, notes, or reference materials.

- `assets`: To store any images, stylesheets, or other static files you might need.

- `scripts`: For any auxiliary scripts or utilities you might write or acquire.

Use Atom's sidebar to right-click on `django_rest_api_project`, choose `New Folder`, and name it accordingly. Repeat this for each sub-directory.

6. Initialize a README File:

It's a good practice to have a `README.md` file at the root of your project. This markdown file will contain information about the project, its purpose, setup instructions, and any other relevant details.

- Right-click on `django_rest_api_project` in Atom.

- Select `New File` and name it `README.md`.

- Open the file and type in a brief introduction to your project. You can expand on this as your project grows.

Conclusion:

Congratulations! You've now set up a structured and organized workspace for your Django REST API project. This organized setup will ensure you can easily manage, update, and collaborate on your project without any hassles. As we move forward, always remember to maintain this structure, as an organized workspace is key to a successful development process.

Creating a Git project

Assets, Resources, and Materials:

- Git: A distributed version control system (VCS). (Get/Acquire: Download and install from [Git's official website](https://git-scm.com/))

- GitHub: A platform for hosting and versioning code. (Get/Acquire: Sign up for a free account at [GitHub](https://github.com/))

Introduction

Creating a Git project means initializing a new repository where your project's files and their history will be stored. This repository acts as a container for your project, tracking the changes you make. Using a VCS like Git is crucial in software development, allowing you to manage versions of your project, collaborate with others, and rollback to previous versions if needed.

Steps to Create a Git Project

1. Install Git

Before initializing a Git project, ensure you've installed Git. If you haven't, refer to Chapter 3 or Chapter 4 (depending on your OS) to install Git.

2. Navigate to Your Workspace

Using your terminal or command prompt, navigate to the directory (folder) where you want your project to reside using the `cd` (change directory) command. For example:

```
cd path/to/your/workspace
```

3. Initialize a New Git Repository

Inside your workspace, run the following command:

```
git init
```

This will initialize a new Git repository and begin tracking an existing directory. You'll see a message like: "Initialized empty Git repository in /path/to/your/workspace/.git/"

4. Add Your Files

Before Git can track your files, you need to add them to the repository. Start by creating a new file or adding existing files.

To add all files in the directory:

```
git add .
```

To add specific files:

```
```

```
git add filename.ext
```

5. Commit Your Files

Committing in Git creates a snapshot of the changes you've made. It's like saving a version of your project. To commit the files you've added, use:

```
git commit -m "Initial commit"
```

Replace "Initial commit" with a brief description of the changes you've made.

6. Create a Remote Repository on GitHub

To push your local repository to GitHub:

- Log into your GitHub account.

- Click the '+' icon in the top right corner and select 'New repository'.

- Fill in a repository name, description, and other settings.

- Click 'Create repository'.

7. Link Local Repository to Remote Repository

In your terminal, add the remote repository:

```
git remote add origin https://github.com/YourUsername/YourRepoName.git
```

Replace 'YourUsername' with your GitHub username and 'YourRepoName' with the name of the repository you just created.

8. Push to GitHub

To upload your local repository to GitHub, use:

“`

git push -u origin master

“`

This will push your code to the master branch of your GitHub repository.

Conclusion

You’ve now successfully created a Git project and connected it with a remote repository on GitHub. By doing this, you not only track changes locally but also have an online backup of your code. As you progress in your project, remember to regularly commit your changes and push them to GitHub to keep both local and remote repositories up to date.

In the next chapter, we’ll look into pushing our project to GitHub, ensuring that all our code is safely stored in an online repository. This will set the foundation for collaboration and sharing your project with others.

Pushing to GitHub

Assets, Resources, and Materials for this Chapter:

1. GitHub Account

    (How to Acquire: Create a free account on [GitHub](https://github.com/). It’s a platform where millions of developers store, share, and collaborate on their code.)

    (Purpose: Required to create repositories and push our code.)

2. Git

    (How to Acquire: Download and install from [Git’s official website](https://git-scm.com/).)

    (Purpose: A version control tool that helps track changes in source code and collaborate with others.)

Introduction

Pushing your project to GitHub not only serves as a backup but also allows other developers to collaborate, view, or even use your project. With the development environment set up in the previous chapters, we are now ready to push our project to GitHub. Let’s dive in!

---

1. Setting Up Your GitHub Repository

1.1 Log into GitHub: Open your browser and log into your GitHub account.

1.2 Create a New Repository: On the top right corner, click on the ‘+’ symbol and choose ‘New repository’. Name your repository, e.g., “django_rest_api_beginner”. Provide a description if desired. For this tutorial, let’s keep the repository public. Do not initialize with a README, .gitignore, or license - we’ll add those manually later.

---

2. Initializing a Local Git Repository

With the GitHub repository set up, it’s time to prepare our local project.

2.1 Open the Terminal or Command Prompt: Navigate to your project directory using the `cd` command. For example:

“`bash

cd path_to_your_project_directory/django_rest_api_project/

“`

2.2 Initialize Git: Run the following command to initialize a new git repository:

“`bash

git init

“`

---

3. Linking Local Repository to GitHub

3.1 Add Remote Repository: To link your local repository to your GitHub repository, use the following command:

“`bash

git remote add origin https://github.com/your_username/django_rest_api_beginner.git

“`

Replace `your_username` with your GitHub username.

---

4. Staging and Committing Changes

Before pushing to GitHub, you need to stage (track) your project files and commit them to your local repository.

4.1 Stage Files: Stage all project files with:

“`bash

git add .

“`

4.2 Commit Files: Commit the staged files with a commit message:

“`bash

git commit -m “Initial commit”

“`

---

5. Pushing to GitHub

5.1 Push: Push the committed changes to your GitHub repository using:

“`bash

git push -u origin master

“`

---

6. Verifying on GitHub

Open your GitHub repository in the browser. You should now see all your project files listed.

---

7. Handling README, .gitignore, and License (Optional)

It’s a good practice to include a `README.md` for project description, `.gitignore` to exclude unnecessary files/folders, and a license file.

7.1 Creating a README.md: In your project root, create a `README.md` file. Edit this file to provide a brief description of your project, how to set it up, etc.

7.2 Setting up .gitignore: Create a `.gitignore` file in your project root. Here, you can specify files or folders you don’t want to track, like:

“`

*.log

*.cache

__pycache__/

“`

7.3 Adding a License: This step is optional, but if you want others to use your code, it’s essential. You can choose a license from [Choose a License] (https://choosealicense.com/) and add it to your project.

Once you’ve added these files, remember to stage, commit, and push them to GitHub.

Conclusion

Congratulations! You’ve successfully pushed your project to GitHub. This not only acts as a version control but also allows other developers to see, use, or contribute to your project. As you make changes to your project, always remember to push those changes to GitHub to keep it updated.

Section 4:

Creating a Development Server

Creating a Vagrant file

Welcome to Chapter 8! In this chapter, we'll dive deep into the process of creating a Vagrantfile. This Vagrantfile will serve as the blueprint for our development server, ensuring that our environment remains consistent and easy to replicate across various machines.

Assets, Resources, and Materials for this Chapter:

- Vagrant: The main software we're working with in this section. If you haven't already installed Vagrant, you can download it from [Vagrant's official website](https://www.vagrantup.com/).

- VirtualBox: Vagrant works hand-in-hand with providers like VirtualBox to manage virtualization. Get it from the [VirtualBox official website](https://www.virtualbox.org/).

- Text Editor: You'll need a text editor to create and modify the Vagrantfile. We recommend Atom, as mentioned earlier in the book. Download Atom [here](https://atom.io/).

Step 1: Initializing a Vagrantfile

1. Open your terminal or command prompt.

2. Navigate to your project's root directory using the `cd` command.

3. Once you're in the project directory, type the following command:

“`bash

vagrant init

“`

Running this command will create a new `Vagrantfile` in your project directory with default settings.

Step 2: Configuring the Vagrantfile

Open the `Vagrantfile` using Atom or your preferred text editor. You'll notice that it's filled with a lot of comments and a basic configuration.

Let's edit this file to suit our needs:

1. Setting the Box: Find the line that looks like:

“`ruby

config.vm.box = “base”

“`

Uncomment it (remove the `#` at the beginning) and change `“base”` to the name of the box you want to use. For this tutorial, we'll use `ubuntu/bionic64`, which is an Ubuntu 18.04 LTS 64-bit box. It should look like this:

“`ruby

config.vm.box = “ubuntu/bionic64”

“`

2. Networking: Allow port forwarding so that our app can be accessed from our host machine. Find and uncomment or add:

“`ruby

config.vm.network “forwarded_port”, guest: 8000, host: 8000

“`

This means the application running on port 8000 in the virtual machine can be accessed on port 8000 on the host machine.

3. Synced Folders: Vagrant syncs the project directory (on your host machine) with the `/vagrant` directory (on the guest machine). This means any changes you make on your host machine will reflect in the VM and vice versa.

“`ruby

config.vm.synced_folder “.”, “/vagrant”

“`

Step 3: Validating the Vagrantfile

After saving the `Vagrantfile`, return to your terminal or command prompt and navigate to the directory containing the `Vagrantfile`.

Run the following command to validate the configuration:

“`bash

vagrant validate

“`

If everything is okay, Vagrant will confirm with a message indicating that the configuration is valid.

Step 4: Booting up the Virtual Machine

Now that our Vagrantfile is set up, it’s time to start the VM with:

“`bash

vagrant up

“`

This command will download the specified box (if it hasn’t been downloaded yet) and start a virtual machine with the configurations we set in the Vagrantfile.

Summary

In this chapter, we successfully created a Vagrantfile tailored to our project's needs, configured it, and used it to boot up our development server. This Vagrant setup ensures that our development environment remains consistent, making it easier for team collaboration and reducing the "it works on my machine" issues.

In the next chapter, we'll dive deeper into configuring our Vagrant box, making it a robust environment for our Django REST API development.

Configuring our Vagrant box

Assets, Resources, and Materials Required for this Chapter:

- Vagrant: This is the tool we're using to manage and provision our virtual development environments. If you haven't already, download and install it from [the Vagrant website](https://www.vagrantup.com/).

- VirtualBox: This software will allow our Vagrant box to run as a virtual machine on our computer. Make sure it's already installed. If not, you can download it from [the VirtualBox website](https://www.virtualbox.org/).

- A Text Editor: We'll be using Atom in this course as mentioned earlier. Ensure Atom (or your preferred editor) is installed and ready for use.

- Vagrant Base Box: This is essentially the operating system image that our Vagrant box will be based on. For this tutorial, we'll use `ubuntu/bionic64`, which is a popular Ubuntu 18.04 LTS image.

Introduction:

Now that we've created a Vagrant file, it's time to configure our Vagrant box. This involves specifying settings for the virtual machine (like memory, CPUs,

network, etc.), as well as any provisioning scripts that set up our development environment.

Step-by-step Configuration:

1. Box Configuration:

Open the Vagrantfile you created in the previous chapter. Start by specifying the base box for our Vagrant box.

```ruby
config.vm.box = "ubuntu/bionic64"
```

2. Setting Machine Resources:

Adjust the virtual machine's resources according to your machine's capabilities. A standard setting might look like this:

```ruby
config.vm.provider "virtualbox" do |v|
    v.memory = "1024"
    v.cpus = 2
end
```

This assigns 1 GB of RAM and 2 CPUs to the virtual machine.

3. Network Configuration:

We can set up a private network for our Vagrant box, which allows us to access it via a private IP address.

```ruby
config.vm.network "private_network", type: "dhcp"
```

With the above configuration, Vagrant will automatically assign an IP address to the machine.

Alternatively, you can specify a static IP:

```ruby
config.vm.network "private_network", ip: "192.168.33.10"
```

4. Folder Syncing:

By default, Vagrant shares your project directory to the `/vagrant` directory in your guest machine. This allows for easy sharing of files between your host machine and your Vagrant virtual machine. You can further customize this if needed:

```ruby
config.vm.synced_folder "src/", "/srv/website"
```

This would sync the `src` directory from your host to the `/srv/website` directory on the guest.

5. Provisioning:

Vagrant allows you to automate the installation and configuration of software on the machine via provisioning scripts. For now, we'll use a simple shell script to update the package manager and install some necessary packages.

Add this to the Vagrantfile:

```ruby
config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install -y python3 python3-pip
SHELL
```

This script will be executed the first time you run `vagrant up`.

6. Additional Plugins:

If there are additional Vagrant plugins you'd like to use, they can be installed via the Vagrant command line and then configured within the Vagrantfile. For instance:

“`bash

$ vagrant plugin install vagrant-vbguest

“`

The `vagrant-vbguest` plugin automatically installs the host's VirtualBox Guest Additions on the guest system.

Starting the Vagrant Box:

Once you've finished configuring the Vagrantfile, save the file and head over to your terminal. Navigate to the directory where your Vagrantfile is located and run:

“`bash

$ vagrant up

“`

This command will read the Vagrantfile, download the `ubuntu/bionic64` image if it's not already downloaded, and start the virtual machine with all the configurations you specified.

Conclusion:

Your Vagrant box is now up and running with the configurations we just set. This will serve as the foundation for our Django development server. With a few simple configurations, we've created a consistent and reproducible development environment, which is crucial for preventing the common “it works on my machine” issue. In the next chapter, we'll delve into running and connecting to our development server.

(Note: Always remember to run `vagrant halt` to shut down the virtual machine when you're done, or `vagrant suspend` if you want to save the current state and

resume later. To destroy the virtual machine completely (and free up disk space), use `vagrant destroy`. You can always rebuild the machine later with `vagrant up`.)

Running and connecting to our dev server

Assets, Resources, and Materials for this chapter:

1. Vagrant: This is our primary tool for managing our virtual development environments. (Acquire it from [Vagrant's official website](https://www.vagrantup.com/))

2. VirtualBox: This is our hypervisor of choice for running our virtual machines. (Available at [VirtualBox's official website](https://www.virtualbox.org/))

3. Terminal or Command Prompt: For running commands.

4. SSH Key Pair: Secure Shell keys that allow us to securely connect to our Vagrant box. (You should have these set up when you installed Vagrant, but we'll briefly review how to ensure they're in place.)

Introduction

After you've set up Vagrant and VirtualBox, the next logical step is to get your development server running and to be able to connect to it. By the end of this chapter, you'll know how to launch your Vagrant environment, SSH into it, and also manage it efficiently.

---

1. Running Your Vagrant Box

Before we can run our Vagrant box, we need to navigate to the directory containing our `Vagrantfile`. The

`Vagrantfile` contains the configuration details for our virtual environment.

“`bash

$ cd path/to/your/vagrant/project

“`

Now, start up the Vagrant box using the following command:

“`bash

$ vagrant up

“`

This command will read the `Vagrantfile` and start up the virtual machine accordingly. This process might take a few minutes the first time, as Vagrant needs to download the necessary box files.

---

2. SSH Into Your Vagrant Box

Once the box is up and running, connecting to it is simple:

“`bash

$ vagrant ssh

“`

This will initiate a secure shell (SSH) session into your Vagrant box. You’re now in a full-fledged Linux environment, separate from your host operating system.

Note: You don’t need to provide any SSH keys or passwords because Vagrant has conveniently set up key-based authentication for you. If you ever need to access these keys, they are typically located in your project directory under `.vagrant/machines/default/virtualbox/private_key`.

---

3. Managing Your Vagrant Environment

Here are some essential commands for managing your Vagrant box:

- Pause your machine: This will save the current running state of the machine.

  “`bash

  $ vagrant suspend

  “`

- Halt or shut down the machine: This will gracefully shut the machine down.

  “`bash

  $ vagrant halt

  “`

- Destroy your machine: This removes all traces of the machine from your system. The next time you run `vagrant up`, it will create a new machine from scratch.

  “`bash

  $ vagrant destroy

  “`

- Checking the status: This will display the status of the machine.

  “`bash

  $ vagrant status

  “`

---

4. Shared Folders

By default, Vagrant shares the project directory (where your `Vagrantfile` is located) to the `/vagrant` directory in your virtual machine. This means you can easily share files between your host and guest machine.

For example: If you have a file named `hello-world.py` in your project directory on your host machine, you’ll find it under `/vagrant/hello-world.py` on your guest machine.

Conclusion

By this point, you've successfully launched and connected to your development server using Vagrant. You have also learned some basic management commands to handle your virtual environment effectively. As you proceed, remember that this environment is isolated; what you do here won't affect your main operating system, giving you the freedom to experiment and learn.

Running a Hello World script

Assets, Resources, and Materials Required:

- Python: If you've followed the previous chapters, you'll have Python installed in your environment. If not, you can download it from the [official Python website](https://www.python.org/downloads/). We'll use Python to run our simple Hello World script.

- Atom: A popular open-source text editor that can be used for coding. You've already installed it in Chapter 3 or 4. If you haven't, download it from the [official Atom website](https://atom.io/).

- Vagrant Development Server: Ensure you have your Vagrant dev server up and running. If not, refer back to the previous chapters in this section.

Introduction:

One of the most traditional ways to start any coding journey is by writing a simple script that displays the message: "Hello, World!" In this chapter, we'll do exactly that, but within the context of our development server. This will provide an introduction to executing Python scripts in our environment.

Steps to Running a Hello World Script on Your Development Server:

1. Access Your Development Server:

If you haven't already, start your Vagrant development server:

```bash
vagrant up
```

Once the server is up, SSH into it:

```bash
vagrant ssh
```

2. Navigate to Your Workspace:

Now, navigate to your workspace directory, the one you created in Chapter 5. This will be our working directory for our script.

```bash
cd /path_to_your_workspace/
```

3. Creating the Script:

Using Atom, create a new file named `hello_world.py` inside your workspace. Remember, you can do this on your main operating system, not inside the Vagrant SSH session.

Open `hello_world.py` in Atom and write the following Python code:

```python
print("Hello, World!")
```

Save and close the file.

4. Running the Script:

Back in your Vagrant SSH session, ensure you're still in your workspace directory. Run the script using Python:

```bash
python hello_world.py
```

You should see the output:

```
Hello, World!
```

Congratulations! You've successfully run a Python script inside your development server.

Understanding the Hello World Script:

At its core, the Hello World script provides an introduction to:

- File Creation & Management: You learned how to create, save, and manage a file within your workspace.

- Python's Print Function: The `print()` function in Python outputs text to the console. It's a crucial function you'll use frequently when debugging and building applications.

- Running Python Scripts: By executing `python hello_world.py`, you invoked the Python interpreter to execute the script, showcasing how Python scripts are executed in general.

Conclusion:

While this chapter provided a basic introduction to running scripts in our development environment, it sets the foundation for more complex tasks ahead. As we progress, we'll be delving deeper into creating more

elaborate applications and functionalities using Python and Django.

Remember, every great journey begins with a simple step, or in the world of programming, a “Hello, World!”

Exercise:

1. Modify the `hello_world.py` script to display your name, e.g., “Hello, [Your Name]!”.

2. Try running other simple Python commands within the script, such as basic arithmetic, to get a feel for executing Python scripts.

Section 5:

Creating a Django App

Create Python Virtual Environment

Assets, Resources, and Materials:

- Python: Before we can create a virtual environment, you’ll need Python installed. You can download the latest version from [Python’s official website](https://www.python.org/downloads/). Python, as we know, is the foundational language we’re using for this course.

- pip (Python Package Installer): Most Python installations come with pip by default. However, if it’s missing, you can [follow these instructions](https://pip.pypa.io/en/stable/installing/). Pip allows us to install Python packages easily.

- virtualenv: This is the primary tool we’ll use to create isolated Python environments. If you don’t have it

installed, don’t worry, we’ll cover that.

Purpose of a Virtual Environment:

A virtual environment is an isolated space on your computer where you can install software and Python packages independently of the system-wide Python installation. This isolation prevents conflicts between versions and ensures a clean, controlled development environment, making it easier to manage project-specific dependencies.

Let’s Get Started:

1. Installing virtualenv:

If you haven’t installed virtualenv yet, you can do so using pip:

```
pip install virtualenv
```

2. Creating a Virtual Environment:

Once you’ve got virtualenv installed, navigate to the directory where you want your virtual environment to live (typically inside your project directory) and run:

```
virtualenv myenv
```

Here, `myenv` is the name of your virtual environment. You can name it anything you like, but keeping it descriptive can be helpful.

3. Activating the Virtual Environment:

- On Windows:

  ```
  myenv\Scripts\activate
```

```

- On macOS and Linux:

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

When the virtual environment is activated, you'll see the name of your virtual environment (in this case, `myenv`) at the beginning of your command line prompt. This indicates that the environment is active, and any Python packages you install while it's active will only be installed in this environment.

4. Deactivating the Virtual Environment:

When you're done working in your virtual environment and want to return to the global Python environment, simply run:

```
deactivate
```

5. Installing Packages in the Virtual Environment:

With the virtual environment active, you can use pip to install packages as you normally would. For example, to install Django:

```
pip install django
```

This will install Django only in the active virtual environment and won't affect the system-wide Python installation.

Why Use Virtual Environments?

Virtual environments are critical when developing multiple projects on the same system, especially when
```

these projects have different dependencies. They allow for:

- Isolation: Ensuring that each project has its own set of dependencies that don't interfere with one another.

- Version Control: Allows for different projects to use different versions of packages, without any conflicts.

- Clean Environment: When you start a new project, you can ensure you're working with a clean slate without any unwanted packages.

- Deployment: Makes it easier to manage dependencies when deploying your application to production.

Recap:

In this chapter, we've learned how to set up a Python virtual environment using virtualenv. We now have a dedicated space for our Django project, where we can manage our dependencies in isolation. In the next chapter, we'll begin installing the necessary Python packages to start our Django project.

Install required Python packages

Assets, Resources, and Materials:

1. Python: Ensure you have Python (preferably version 3.7 or higher) installed on your development server. To check, run `python —version` or `python3 —version` in the terminal. (You can download and install Python from [python.org](https://www.python.org/downloads/))

2. pip: Python's package installer. It's typically installed with Python. To check its installation, run `pip —version` or `pip3 —version` in the terminal. (If not installed, [follow these instructions](https://pip.pypa.io/en/stable/installing/)).

3. Virtual Environment: Though not a strict requirement, using a virtual environment for your project ensures that dependencies do not clash between projects. (Installation instructions provided below).

Introduction:

To begin building our Django application, we first need to ensure that we have all the necessary Python packages installed. These packages will facilitate the creation, management, and enhancement of our Django app. In this chapter, we'll walk through the process of installing the necessary Python packages.

Step 1: Setting Up a Virtual Environment

A virtual environment is an isolated environment where you can install packages without affecting the global Python environment or other projects. It's a best practice to use virtual environments for each of your Python projects.

a. To install the virtual environment package, run:

“`bash

pip install virtualenv

“`

b. Navigate to your project directory (or where you want to create the virtual environment) and run:

“`bash

virtualenv venv

“`

c. Activate the virtual environment. The activation command varies depending on your operating system:

- Windows:

“`bash

venv\Scripts\activate

“`

- macOS and Linux:

“`bash

source venv/bin/activate

“`

Once activated, you should see `(venv)` at the beginning of your terminal prompt, indicating that you are now working within the virtual environment.

Step 2: Installing Django and Django REST Framework

With our virtual environment active, we can proceed to install Django and the Django REST Framework.

a. Install Django by running:

“`bash

pip install django==2.2

“`

By specifying `==2.2`, we’re ensuring that we install version 2.2 of Django, which our course is based on.

b. Next, install the Django REST Framework:

“`bash

pip install djangorestframework==3.9

“`

Step 3: Verifying the Installations

After installing, it’s a good practice to verify that everything was installed correctly.

a. Check the installed Django version with:

“`bash

django-admin —version

“`

This should display `2.2`.

b. To check the installed packages in the virtual environment, run:

“`bash

pip freeze

“`

This will list all installed packages, and you should see both `Django==2.2` and `djangorestframework==3.9` among them.

Conclusion:

Congratulations! You’ve successfully set up a virtual environment and installed the necessary Python packages for building your Django app. With these tools in place, you’re now ready to dive deeper into creating your Django project and application in the subsequent chapters.

Remember, once you’ve finished working on your project for the time being, you can deactivate the virtual environment by simply running:

“`bash

deactivate

“`

In the next chapter, we’ll explore how to initialize a new Django project and create our first Django app.

Remember to always consult the official documentation for [Django](https://docs.djangoproject.com/en/2.2/) and [Django REST Framework](https://www.django-rest-framework.org/) for any additional details or updates related to these packages.

Create a new Django project & app

Assets, Resources, and Materials:

- Python: Ensure you have Python installed. (Download from [Python's official website](https://www.python.org/downloads/))

- Django: We will install this in this chapter using pip.

- Command Line Interface (CLI): Access via Terminal (on macOS/Linux) or Command Prompt/PowerShell (on Windows).

- Text Editor: Atom (or any preferred text editor)

Introduction:

In this chapter, we will walk you through the process of creating a new Django project, followed by creating an app within that project. A Django project is essentially a collection of settings, configurations, and apps. An app, on the other hand, is a self-contained module that can represent anything – from a blog to a user authentication system. Each app typically has its models, views, and controllers.

Step 1: Install Django

Before we begin creating our project, we need to ensure Django is installed. If not, let's install it using `pip`, Python's package manager.

"`bash

pip install django==2.2

"`

Step 2: Start a New Django Project

Once Django is installed, we can now create a new project. Navigate to the directory where you want your project to live, and then run the following command:

“`bash

django-admin startproject myproject

“`

Replace `myproject` with your desired project name. This command will create a new directory with the name `myproject` containing all the necessary files for a Django project.

Step 3: Start a New Django App

Now that our project is created, let’s navigate into the project directory:

“`bash

cd myproject

“`

From within the project directory, we can create our app. Let’s say we’re building a blog; we might call our app ‘blog’. To create this app, use the command:

“`bash

python manage.py startapp blog

“`

This command will generate a new directory called `blog` inside your project directory, with the structure for a new app.

Step 4: Explore the Project & App Structure

Now, if you navigate to your project directory using a file explorer or your text editor (Atom), you’ll see the following structure:

“`

```
myproject/
│
├── manage.py
│
├── myproject/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│
└── blog/
    ├── migrations/
    │   └── __init__.py
    ├── admin.py
    ├── apps.py
    ├── models.py
    ├── tests.py
    └── views.py
```

- manage.py: This is a command-line utility to interact with your project.

- myproject/settings.py: This file contains settings for your project.

- myproject/urls.py: This file will contain all the URL patterns for your project.

- blog/models.py: This is where you'll define the data models for your app.

- blog/views.py: Here, you'll define the logic and control flow for handling requests and responses.

Step 5: Register the App with the Project

For Django to recognize our new ‘blog’ app, we need to register it. Open `myproject/settings.py` in Atom (or your preferred text editor) and locate the `INSTALLED_APPS` setting. Add the name of your app to this list:

“`python

INSTALLED_APPS = [

    # … other default apps here …

    ‘blog’,

]

“`

Conclusion:

Congratulations! You’ve now set up a new Django project and created your first app within it.

Remember, Django follows the DRY (Don’t Repeat Yourself) principle. With just a few commands, we have set up a robust structure that will save us time and effort as we progress with our project.

Enable our app in the Django settings file

In this chapter, we’ll walk through the process of enabling our newly created Django app in the Django settings file, ensuring that it is acknowledged and managed by the Django framework. This step is vital for integrating any new app into a Django project.

Assets, Resources, and Materials for this chapter:

- Django: Our web framework of choice. (Already installed in the previous chapters. If not, revisit Chapter 14.)

- Django Project: The overarching project folder containing all of your Django apps and configurations.

- Django App: The app you've just created. (If not yet created, please refer to Chapter 14.)

- Text Editor (Atom): To edit the Django settings file. (If not installed, refer to Chapter 3 or 4 for installation steps based on your operating system.)

Step-by-step guide to enable your Django app:

1. Locate Your Django Settings File

   - Open the Django project in your text editor (Atom).

   - Navigate to the main project folder (this is the folder named after your project, not the one named after your app).

   - Inside this folder, you will find a file named `settings.py`. This file contains configurations for your entire Django project.

2. Open the `settings.py` File

   - Double-click on the `settings.py` file to open it in your text editor.

3. Locate the `INSTALLED_APPS` Setting

   - Scroll down or search for `INSTALLED_APPS`. This is a list in Django that contains all the apps that are currently enabled and recognized by your Django project.

4. Add Your App to the List

   - At the end of the `INSTALLED_APPS` list, add a new line with the name of your Django app enclosed in single or double quotes.

   "`python

   INSTALLED_APPS = [

       … # other pre-existing apps here

```
    'your_app_name',
]
```

Replace ``'your_app_name'`` with the name of your app.

5. Save the File

- Once you've added your app name to the list, save the `settings.py` file.

6. Verify Your Changes

- To ensure that Django acknowledges your app, run the following command in your terminal or command prompt:

```
python manage.py check
```

This command will perform a system check. If there are no issues, it should return no errors.

7. Understanding the Purpose

- Why did we do this? `INSTALLED_APPS` tells Django which applications are active for this project and should be taken into account for various operations like database migrations, admin interface generation, etc. Any time you create a new Django app, you need to make sure it's listed in the `INSTALLED_APPS` setting so Django knows about it.

Summary:

By following these steps, you've integrated your app into the Django project. It's crucial to ensure that every app you create is added to the `INSTALLED_APPS` list, so Django can manage it correctly. With your app now officially recognized by Django, you can begin defining

models, views, and other functionalities specific to your app in the coming chapters.

In the next chapter, we will start writing tests to ensure our app functions as expected and commit our changes using Git.

Test and commit our changes

Assets, Resources, and Materials required for this chapter:

- Terminal or Command Prompt (pre-installed in most operating systems)

- Python (already installed, see Chapter 12)

- Django Framework (installed in Chapter 14)

- Git (installed in Chapter 3 or 4 based on OS)

- Atom Editor (installed in Chapter 3 or 4 based on OS)

- GitHub Account (created in Chapter 7 for pushing the project)

Introduction:

After creating your Django project and app, it’s vital to ensure that everything is working as expected. This chapter walks you through how to test your app and subsequently commit your changes using Git.

---

1. Testing Your Django App

Before we commit any code, let’s ensure that everything is working fine. Django comes with a built-in server for local development, which is perfect for our needs.

Step 1: Start the Django development server. Open your terminal or command prompt, navigate to your project directory, and type:

“`bash

python manage.py runserver

“`

Step 2: Once the server starts, open your preferred web browser and go to ` http://127.0.0.1:8000/ `. You should see the Django welcome page, indicating that your project is set up correctly.

Step 3: Check for any possible issues using Django’s built-in check command:

“`bash

python manage.py check

“`

This command checks your entire Django project for common problems. If everything is fine, you’ll see the message “No issues found.”

---

2. Committing Your Changes with Git

Now that we’ve verified our app is working, it’s time to commit our changes.

Step 1: First, open your terminal or command prompt in your project directory.

Step 2: Check the status of your git repository:

“`bash

git status

“`

This command shows the changes you’ve made since the last commit. You should see the files related to the Django project and app that you’ve created.

Step 3: Add the changed files to the staging area:

“`bash

git add .

“`

The `.` after `add` means you’re adding all changes in the current directory (and its subdirectories) to the staging area.

Step 4: Commit the changes:

“`bash

git commit -m “Created Django project and app”

“`

This command creates a new commit with the changes in the staging area. The `-m` flag allows you to add a message describing the commit, which is useful for tracking changes and understanding the project’s history.

Step 5: Push the changes to GitHub:

Ensure that you’ve already set up a remote repository on GitHub (see Chapter 7). Push the changes using:

“`bash

git push origin master

“`

`origin` refers to the default name of the remote repository you’ve set up, and `master` is the main branch of your project.

Conclusion:

You’ve successfully tested your Django app and committed your changes using Git. Regularly testing and committing your changes ensures that you have a working version of your app at all times and that your progress is saved on a platform like GitHub. This habit is crucial in the development world and keeps your code safe and accessible from anywhere.

Section 6:

Setup the Database

What are Django Models?

Assets, Resources, and Materials:

- Django (How to acquire: Install via pip with `pip install django`. Purpose: The web framework we're working with.)

- Python (How to acquire: Download from [Python's official website](https://www.python.org/downloads/). Purpose: The programming language in which Django is written.)

- Django Documentation on Models (How to acquire: Available online at [Django's official documentation site](https://docs.djangoproject.com/en/2.2/topics/db/models/). Purpose: Comprehensive resource on Django models.)

Introduction

In the world of web applications, data is king. Whether you're tracking user profiles, product inventories, or any other kind of information, that data needs to be stored somewhere. In most modern web applications, this "somewhere" is a relational database. And to interact with this database in an organized, structured, and efficient manner, Django provides us with a powerful feature known as "Models".

What is a Django Model?

A Django model is a single, definitive source of truth about your data. It contains the essential fields and behaviors of the data you're storing. Essentially, Django models are a way to define the structure and behavior of your application's data.

Each model maps to a single database table and can be thought of as a Python class that subclasses `django.db.models.Model`. Each attribute of this class represents a field in the database table.

Why Use Django Models?

1. Abstraction: Instead of writing raw SQL queries, you can leverage Django's Object-Relational Mapping (ORM) to interact with your data in a more Pythonic way.

2. Efficiency: Django takes care of creating, reading, updating, and deleting records in the database for you, minimizing the chance for manual error.

3. Consistency: By defining data structures through models, you ensure that your data adheres to a specific format or schema.

4. Validation: Models allow for data validation, ensuring that the data in your database is clean and reliable.

5. Query API: Django models come with a built-in query API that lets you retrieve and manipulate your data in sophisticated ways.

Defining a Simple Django Model

Let's take an example of a simple `Blog` application where you want to store information about `Post`.

“`python

from django.db import models

class Post(models.Model):

    title = models.CharField(max_length=200)

    content = models.TextField()

    pub_date = models.DateTimeField('date published')

“`

In this example:

- `Post` is our Django model which corresponds to a table in our database.

- `title` is a field that allows up to 200 characters.

- `content` is a field that allows for an unlimited number of characters.

- `pub_date` is a date-time field that will store when the post was published.

Once defined, this model gives Django all the information it needs to create a corresponding database table.

Migrations: Reflecting Model Changes in the Database

Django comes with a built-in migrations framework that tracks changes to your models over time. When you create or update a model, Django can automatically generate the SQL needed to make corresponding changes in the database. This allows for a smooth and controlled transition as your application's data structure evolves.

Conclusion

Django models are a cornerstone of any Django web application. They define the shape and behavior of your application's data and come packed with a host of features that make it easy and efficient to work with relational databases. In subsequent chapters, we'll delve deeper into defining custom behavior for our models, relationships between different models, and advanced querying techniques.

Create our User Database Model

Assets, Resources, and Materials:

- Django: The primary web framework used in this course. [Get Django here](https://www.djangoproject.com/download/).

- Python: The programming language used. Ensure you have Python 3.7 or newer. [Download Python here](https://www.python.org/downloads/).

- Django Documentation: Your go-to place for any clarifications or additional information. [Check out Django docs](https://docs.djangoproject.com/).

- A code editor: Such as Atom, as mentioned in the earlier chapters.

Introduction:

In this chapter, we will focus on creating a user database model using Django's Object-Relational Mapping (ORM) system. Models in Django are a way to represent database tables. By defining a User model, you're essentially outlining the schema for the user table in your database.

---

1. Understanding Django Models:

Django models are the single, definitive source of information about your data. They contain the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

Each model is a Python class that subclasses `django.db.models.Model`. The attributes of the model represent database fields.

---

2. Create our User Model:

For our REST API, we want to create a custom user model that can be extended in the future if needed. Django does come with a built-in user model, but by creating a custom one, you gain more flexibility.

Here's a step-by-step guide:

Step 1: Create a new file named `models.py` in your Django app directory (if it doesn't already exist).

Step 2: Open `models.py` and start by importing the necessary modules:

“`python

from django.db import models

from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, PermissionsMixin

“`

Step 3: Define the custom user manager:

Before we can create our custom user model, we need to define a manager for it. This manager will contain helper functions like `create_user` and `create_superuser`.

“`python

class UserManager(BaseUserManager):

    def create_user(self, email, password=None, extra_fields):

        if not email:

         raise ValueError(“The Email field must be set”)

        email = self.normalize_email(email)

        user = self.model(email=email, extra_fields)

        user.set_password(password)

        user.save(using=self._db)

        return user

    def create_superuser(self, email, password=None, extra_fields):

        extra_fields.setdefault(‘is_staff’, True)

        extra_fields.setdefault(‘is_superuser’, True)

        return self.create_user(email, password, extra_fields)

“`

Step 4: Define the `User` model:

Now, let’s define our `User` model which will inherit from `AbstractBaseUser` and `PermissionsMixin`.

“`python

class User(AbstractBaseUser, PermissionsMixin):

    email = models.EmailField(unique=True)

    first_name = models.CharField(max_length=30, blank=True)

    last_name = models.CharField(max_length=30, blank=True)

    date_joined = models.DateTimeField(auto_now_add=True)

    is_active = models.BooleanField(default=True)

    is_staff = models.BooleanField(default=False)

    objects = UserManager()

    USERNAME_FIELD = ‘email’

    REQUIRED_FIELDS = []

    def __str__(self):

        return self.email

“`

In the above code:

- We’ve defined essential fields for our user like `email`, `first_name`, `last_name`, etc.

- We’ve set the `USERNAME_FIELD` to email, which will be used as the unique identifier for authentication.

- The `REQUIRED_FIELDS` list indicates which fields must be provided when creating a user through the command line. Since email is our USERNAME_FIELD, it doesn’t need to be included in this list.

Conclusion:

With the model now defined, you've effectively set the foundation for your user database. The next steps would involve creating migrations (which we will tackle in upcoming chapters) to apply this model to the actual database and subsequently utilize Django Admin to manage these users.

Add a user model manager

Assets, Resources, and Materials Required:

- Python: The main programming language we'll be using. (Acquired via the official [Python website](https://www.python.org/downloads/))

- Django: The web framework we are using to build our backend. (Install via pip: `pip install django`)

- Django's UserManager: A built-in manager provided by Django to handle user-related tasks. (Comes with Django's auth framework)

- Text Editor/IDE: Atom, Visual Studio Code, PyCharm, or any other preferred editor to write our code. (For our example, we're using Atom which can be downloaded from the official [Atom website](https://atom.io/))

- Command Line Terminal: The default terminal/command prompt of your operating system will suffice.

Introduction

As we delve further into setting up our database, one crucial aspect we need to consider is the management of our user model. Django offers a default `UserManager`, but as we'll be using a custom user model, we need to set up our own user model manager.

A model manager is essentially a class that contains functions that help in creating and managing objects of

the associated model. In the context of a user model, this can mean functions to create a user, create a superuser, check user credentials, and so on.

Steps to Add a User Model Manager

1. Importing Necessary Modules and Libraries:

   Open your models.py file where you've defined your user model. At the top of this file, you'll need to import a few modules:

   "`python

   from django.contrib.auth.models import BaseUserManager

   "`

2. Defining the Custom User Manager:

   Below your User model, define your custom user manager class. This class will inherit from `BaseUserManager`.

   "`python

   class CustomUserManager(BaseUserManager):

       pass

   "`

3. Adding Methods to the Manager:

   Within the `CustomUserManager` class, we'll add methods that will assist in creating users and superusers.

   "`python

   def create_user(self, email, password=None, extra_fields):

       """

       Create and return a User with the given email and password.

       """

```
        if not email:
            raise ValueError("Users must have an email address")
        email = self.normalize_email(email)
        user = self.model(email=email, extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user
    def create_superuser(self, email, password=None, extra_fields):
        """
        Create and return a superuser with the given email and password.
        """
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser must have is_staff=True.')
        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser must have is_superuser=True.')
        return self.create_user(email, password, extra_fields)
    ```
```

4. Link the Custom User Manager to the User Model:

Now that we've created our custom user manager, we need to link it with our user model. Within the user model, create an instance of the custom user manager.

```python
```
```

```
    class CustomUser(AbstractBaseUser,
PermissionsMixin):
        …
        objects = CustomUserManager()
    “`
```

The `objects` attribute allows us to call our custom user manager methods using the model.

Explanation:

- The `create_user` method is for creating regular users. It takes in an email (which we’re using as a primary means of identification) and a password, along with any other fields that you might have defined in your user model.

- The `create_superuser` method is for creating admin users. It ensures that the `is_staff` and `is_superuser` fields are set to True, a requirement for users that need access to the Django admin site.

- The `normalize_email` function is a helper function provided by Django that normalizes the domain part of the email, converting it to lowercase. This ensures consistency in the database.

Conclusion

With the user model manager in place, our custom user model is now equipped to handle user creation and management tasks. This lays down the foundation for managing users in our REST API. As we move forward, you’ll see the true power and flexibility of having a custom user model and manager in Django, allowing you to customize user behavior as per the requirements of your application.

(Note: This chapter provides the basic implementation of a custom user model manager. Depending on the

requirements of your project, you might need to add more methods or adjust the existing ones to fit your needs.)

Set our custom user model

Assets, Resources, and Materials for this Chapter:

1. Django: (Acquired from [Django's Official Website](https://www.djangoproject.com/download/)). The web framework we are using to build our application.

2. Text Editor (e.g., Atom): (Download from [Atom's Official Website](https://atom.io/)). Where we'll write our code.

3. Python Environment: Ensures you have Python installed and can run Django. (If not already set up, refer to Chapter 12).

4. Previous User Model: Created in Chapter 18. We will be modifying and extending this model.

5. Django Documentation: (Available at [Django's Official Documentation](https://docs.djangoproject.com/)). Helpful for any additional details or clarifications.

Introduction:

Django comes with a built-in user model for authentication. However, often we want to extend this model or entirely replace it to cater to our application needs. In this chapter, we will set our custom user model to serve as the default user model for authentication and user-related operations.

Steps:

1. Review Existing User Model:

First, navigate to the user model you created in Chapter 18. This is the foundation we'll be building upon.

2. Define the Custom User Model:

In your models.py file (within the app where your user model resides), start by importing the necessary modules:

```python
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, PermissionsMixin
from django.db import models
```

Extend the AbstractBaseUser and PermissionsMixin for our custom user model. Here's an example that builds upon the user model you might have in place:

```python
class CustomUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name']
    objects = CustomUserManager()
```

Here, we are making the email as the USERNAME_FIELD which means users will use their email to log in instead of a username.

3. Create a Custom Manager for the User Model:

The CustomUser model requires a manager to handle its database operations. Extend BaseUserManager to create our custom manager:

“`python

class CustomUserManager(BaseUserManager):

def create_user(self, email, first_name, last_name, password=None, extra_fields):

if not email:

raise ValueError(“Email field is required”)

email = self.normalize_email(email)

user = self.model(email=email, first_name=first_name, last_name=last_name, extra_fields)

user.set_password(password)

user.save(using=self._db)

return user

def create_superuser(self, email, first_name, last_name, password=None, extra_fields):

extra_fields.setdefault(‘is_staff’, True)

extra_fields.setdefault(‘is_superuser’, True)

return self.create_user(email, first_name, last_name, password, extra_fields)

“`

4. Update Django Settings:

Now that our custom user model is set up, we need to inform Django to use it as the default user model. In your project’s settings.py file, add or update the following line:

“`python

AUTH_USER_MODEL = ‘your_app_name.CustomUser’

“`

Replace `your_app_name` with the name of the app where your CustomUser model resides.

5. Migrations:

Since we've made changes to our model, we need to create migrations and apply them. This can be done with:

“`bash

python manage.py makemigrations

python manage.py migrate

“`

6. Testing:

It's essential to test our custom user model to ensure everything works as expected. You can do this by creating a new user through the Django admin site or using Django's shell. If everything's set up correctly, there shouldn't be any errors, and the user data should get saved in the database.

Conclusion:

Customizing the user model in Django allows developers to tailor the authentication system according to the needs of the application. By setting our custom user model, we have more control over user data, how it's stored, and the fields it should contain.

Create migrations and sync DB

Assets, Resources, and Materials needed:

- Python (To run Django commands) _(Already installed in previous chapters)_

- Django project setup _(Covered in previous chapters)_

- Terminal or Command Prompt

- Code Editor (like Atom) _(Used in previous chapters)_

---

1. Introduction

Django’s database-abstraction API allows you to create, retrieve, update, and delete records without writing a single line of SQL. This is possible due to its Object-Relational Mapping (ORM) layer. One of the most powerful parts of Django is its ORM, and migrations play a key role in it. Migrations are Django’s way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema.

---

2. Why Migrations?

When we define or modify our model classes in Django, the actual database tables aren’t immediately affected. Instead, Django tracks the changes and allows us to apply them in a systematic way using migrations. Migrations solve the problem of evolving a database schema without manual intervention.

---

3. The `makemigrations` Command

Before we can apply migrations, we need to create them. After making changes to a Django model:

1. Open your terminal or command prompt.

2. Navigate to the root directory of your Django project.

3. Enter the following command:

“`bash

python manage.py makemigrations

“`

This will check your model definitions against the current state of your database and generate migration files – scripts that, when run, will modify your database schema to reflect the changes you’ve made to your models.

You should see an output similar to:

“`bash

Migrations for ‘app_name’:

  app_name/migrations/0002_auto_YYYYMMDD_HHMM.py

    - Create model YourModelName

“`

---

4. Checking SQL Code for Migrations

If you want to see the SQL code that Django has generated for a particular migration, you can use the following command:

“`bash

python manage.py sqlmigrate app_name 0002_auto_YYYYMMDD_HHMM

“`

Replace `app_name` with the name of your app and `0002_auto_YYYYMMDD_HHMM` with the name of the migration file you want to inspect. This will display the SQL statements that will be run when this migration is applied.

---

5. Applying Migrations

To apply the migrations and sync your database, use the following command:

“`bash

python manage.py migrate

“`

This command looks at the `INSTALLED_APPS` setting, finds migrations that haven’t been applied yet, and runs them against your database in the correct order.

You should see output indicating which migrations are being applied and if they were successful.

---

6. Rolling Back Migrations

In case you made a mistake or want to revert to a previous database state, Django allows you to “unapply” migrations. To roll back a migration:

“`bash

python manage.py migrate app_name 0001_initial

“`

This will unapply all migrations for `app_name` after `0001_initial`.

---

7. Common Issues and Tips

- No changes detected: If you run `makemigrations` and see “No changes detected”, double-check that you’ve saved your models.py file and that the model’s app is included in the `INSTALLED_APPS` setting.

- Dependencies: Migrations can have dependencies on other migrations or even other apps. Django is usually smart about figuring out the correct order, but sometimes you might need to give it hints.

- Conflicting migrations: If you’re working in a team and multiple developers are making changes to the same models, you might encounter “conflicting migrations”. You’ll need to coordinate with your team to resolve these manually.

---

8. Summary

In this chapter, we delved into the world of Django’s database migrations. Migrations allow us to evolve our database schema in a controlled and systematic manner. By using the commands `makemigrations` and `migrate`, we can efficiently handle changes to our models and keep our database in sync. Remember to always test migrations in a development environment before applying them in production to avoid unexpected issues.

Section 7:

Setup Django Admin

Creating a superuser

Objective: By the end of this chapter, you should be able to create a superuser account for your Django project which gives you access to the Django Admin interface.

Assets and Materials:

1. Command Line Interface (CLI): Most operating systems have this pre-installed. We'll use this to run commands that will create the superuser.

    - How to acquire: Pre-installed on most operating systems.

    - Use: To run various commands.

2. Django Project: You should already have this set up from the previous chapters.

    - How to acquire: If you've been following along, you should have already set this up. If not, refer to the previous chapters.

    - Use: The superuser will be associated with this Django project.

3. Django's manage.py script: This comes pre-packaged with every Django project.

    - How to acquire: Automatically generated when you create a new Django project.

    - Use: To manage various aspects of the Django project including creating superusers.

Introduction

In Django, the term “superuser” refers to a user who has all permissions and unrestricted access to the Django Admin interface. The Django Admin is a powerful built-in tool that allows you to manage the data in your app with ease. However, in order to access it, you need a user account with the necessary permissions - the superuser.

Steps to Create a Superuser:

1. Open Your Command Line Interface (CLI):

   Navigate to the directory where your Django project is located. Make sure you activate your virtual environment if you’ve set one up.

2. Run the `createsuperuser` Command:

   In the root directory of your Django project, where the `manage.py` file is located, run the following command:

   “`bash

   python manage.py createsuperuser

   “`

   This command tells Django to start the process of creating a new superuser.

3. Fill Out the Required Information:

   After running the command, you’ll be prompted to enter details for the superuser:

   - Username: This will be used to log into the Django Admin. Choose something you’ll remember.

   - Email Address: A valid email address. This can be useful for password recovery and notifications.

   - Password: Choose a strong password. You’ll need to enter it twice to confirm.

   Note : If any of the information you enter conflicts with existing data (for example, an existing user with the same username or email address), Django will alert you and ask you to correct it.

4. Confirmation:

Once you've successfully filled out the required information and there are no conflicts, you should see a success message confirming the creation of the superuser.

Accessing the Django Admin:

Now that you've created a superuser, you can access the Django Admin interface. Start your Django server with the following command if it's not already running:

"`bash

python manage.py runserver

"`

Open a web browser and navigate to ` http://127.0.0.1:8000/admin/ ` or the respective address you've set up for your project. Here, you can log in using the superuser credentials you just created.

Conclusion:

Congratulations! You've successfully created a superuser for your Django project. With this account, you can manage all aspects of your application through the Django Admin interface. This capability will prove invaluable as you continue to develop your application.

Note: Remember, the superuser account has complete access to the Django Admin and can make significant changes to the application and its data. Always be cautious while using the superuser account, and consider creating additional users with limited permissions for day-to-day tasks.

Enable Django Admin

Assets, Resources, and Materials:

- Django Framework (To install: `pip install django==2.2`)

- Django Admin Site (Built into Django 2.2; no additional installation required)

- A Django project and app (Refer to previous chapters for creating these)

Introduction:

Django Admin is a powerful and production-ready administrative interface that Django provides out of the box. It lets developers and site administrators create, read, update, and delete records in their database with ease. In this chapter, we will learn how to enable and customize the Django Admin site for our project.

Steps to Enable Django Admin:

1. Ensure ‘django.contrib.admin’ is in `INSTALLED_APPS`:

Open the `settings.py` file of your Django project. You should find a list named `INSTALLED_APPS`. By default, ‘django.contrib.admin’ should already be a part of this list:

“`python

INSTALLED_APPS = [

…

‘django.contrib.admin’,

…

]

“`

If it’s not present, add it.

2. Ensure Middleware is Set Up:

The admin site requires certain middleware classes to function correctly. Ensure your `MIDDLEWARE` setting in `settings.py` includes:

```python
MIDDLEWARE = [
    …
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    …
]
```

3. Set Up the URLs:

To make the admin site accessible via a URL, you need to include its URLs in your project's `urls.py` file.

First, ensure you've imported `admin` and `include` at the top of the file:

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

Then, add the following path to the `urlpatterns` list:

```python
urlpatterns = [
    …
    path('admin/', admin.site.urls),
    …
]
```

This configuration will make the admin site accessible at `/admin/` on your website.

4. Run Migrations:

Ensure you've applied the latest migrations, as Django Admin uses some built-in models. Run:

```bash
python manage.py migrate
```

5. Accessing the Admin Site:

Start your development server:

```bash
python manage.py runserver
```

Now, navigate to `http://127.0.0.1:8000/admin/` in your browser. You should see the Django Admin login page.

Customizing the Admin Interface:

While the default configuration of the Django Admin site is functional, there's much you can do to customize its appearance and behavior:

1. Branding:

Change the site header, title, and index title:

```python
admin.site.site_header = "MyApp Administration"
admin.site.site_title = "MyApp Admin Portal"
admin.site.index_title = "Welcome to MyApp Admin"
```

Place the above code in your app's `admin.py` file.

2. Registering Models:

If you have built models (like our User Database Model from Chapter 18), you can register them with the admin site to manage their records:

“`python

from .models import UserProfile

admin.site.register(UserProfile)

“`

Add this code to your app’s `admin.py` file, and you’ll be able to manage `UserProfile` records from the admin interface.

Conclusion:

The Django Admin site is a powerful tool for managing your application’s data. By enabling and customizing it as shown above, you have given yourself a robust interface to manage the backend data of your application.

Test Django Admin

Assets, Resources, and Materials Required:

- Django project setup from previous chapters.

- Django’s built-in development server.

- Web browser (e.g., Google Chrome, Mozilla Firefox).

- The superuser account created in Chapter 22.

- *(Optional)* Sample data for the user model we’ve created.

Purpose: The Django Admin site provides a web-based interface to manage your application data. Testing the Django Admin ensures that you’ve set it up correctly and that you can easily interact with the data models of your application.

---

1. Introduction to Django Admin Testing

The Django Admin site is one of Django’s most celebrated features. It offers a quick way to create a

management console for your application data without any additional coding. Testing ensures you've set it up correctly, and it functions as expected.

---

2. Starting the Django Development Server

Before you can test the Django Admin, you must ensure your development server is running:

"`bash

python manage.py runserver

"`

Upon running this command, your server will start, and you'll see an output indicating the IP address and port where your server is running, typically ` http://127.0.0.1:8000/ `.

---

3. Accessing the Django Admin Interface

Open your web browser and navigate to the admin page by appending `/admin` to your server's address: ` http://127.0.0.1:8000/admin `

You'll be presented with a login page. Enter the superuser's credentials you created in Chapter 22.

---

4. Navigating the Admin Dashboard

Upon successful login, you'll land on the Django Admin dashboard. Here, you'll see groups representing each app in your project and listings of database models you've registered with the admin interface. If you followed along with the previous chapters, you should see our user model listed.

---

5. Adding and Editing Data

1. Creating a New User Record:

   - Click on the `Users` or the appropriate model name.

- Select “Add User” or a similar option from the top right.

- Fill out the form with the user details.

- Click “Save” at the bottom.

2. Editing an Existing Record:

- From the list of users, click on a user name.

- Modify any fields as needed.

- Click “Save” at the bottom.

---

6. Filtering and Searching Records

On the user list page, you’ll notice a right sidebar with filter options. This allows you to quickly filter records based on certain criteria. Additionally, at the top, there’s a search bar. Try searching for a username or email to see how the search functionality works.

---

7. Deleting Records

To delete a record:

- From the list of users, check the box next to the user(s) you want to delete.

- From the “Action” dropdown menu at the top, choose “Delete selected users” and click “Go”.

- Confirm the deletion on the next page.

---

8. Logging Out of Django Admin

Always ensure you log out of the Admin interface when done, especially if you’re on a public or shared machine. Click on the “Logout” link at the top right corner of the dashboard.

---

9. Ensuring Security

A word of caution: The Django Admin site is powerful, and you should be cautious about who has access to it. In a real-world deployment, consider:

- Using strong, unique passwords for superusers.

- Limiting access to the admin interface by IP.

- Regularly reviewing and updating user permissions.

---

10. Conclusion

The Django Admin interface provides a robust way to manage application data. By thoroughly testing its functionalities, you ensure that data management tasks can be performed easily and efficiently.

Remember, the Django Admin is just one tool in your Django toolbox. While it's fantastic for quick data management tasks, you may often need more customized solutions, which we'll explore in later chapters.

Section 8:

Introduction to API Views

What is an APIView?

Assets, Resources, and Materials:

- Official Django REST Framework Documentation: The official documentation is a comprehensive resource that provides an in-depth understanding of various components including APIView. It can be accessed [here](https://www.django-rest-framework.org/).

- Python: The primary language we will be using. If not already installed, you can download and install Python

from the [official website](https://www.python.org/).

- Django REST Framework: A powerful and flexible toolkit for building Web APIs. You should have this installed as it's required for our course. Instructions for installation can be found in Chapter 13.

Introduction

Before we dive into APIViews and its wonders, it's essential to clarify the fundamental concept of a "View" in Django. In Django, a view is simply a Python function (or a class) that takes in a web request and returns a web response. This response could be an HTML page, a redirect, a 404 error, or even a JSON response in the context of an API. With the Django REST Framework (DRF), this concept gets a boost, making it even more powerful and flexible to work with APIs.

APIView: Elevating Django Views for APIs

APIView is a class-based view provided by the Django REST Framework, specifically tailored for building APIs. While Django's traditional views focus on handling HTML responses and managing forms, APIView focuses on returning structured data like JSON or XML.

Key Features of APIView:

1. HTTP Method Handling: Instead of defining separate views for different HTTP methods (GET, POST, PUT, etc.), with APIView, you can handle these methods by defining corresponding methods on your view class (`get()`, `post()`, and so on).

2. Request Parsing: APIView transforms the incoming HTTP request into a more Django-friendly format. The incoming request data is parsed into Python data types, which simplifies the data handling process.

3. Response Wrapping: It provides a `Response` object, ensuring the data you send back is rendered into the appropriate content type (like JSON).

4. Exception Handling: Instead of handling exceptions manually, APIView provides built-in exception handling that translates many Python exceptions to appropriate HTTP responses.

5. Authentication: You can easily add authentication to your views, ensuring that only authorized users can interact with your API endpoints.

6. Permission Handling: APIView provides mechanisms to define who can do what. For instance, you can specify that only authenticated users can post data, but anyone can read.

7. Content Negotiation: Determines the best response format based on client requests. For instance, while your API might support both JSON and XML, it can use content negotiation to determine which format to send back, based on the client's request headers.

Why Use APIView?

Considering Django already has a robust system for creating views, one might wonder about the necessity of APIView. Here's why:

- Structured Data: As our goal is to build an API, we need to work extensively with structured data (like JSON). APIView simplifies this process.

- Reusability: With class-based views, you can reuse common functionalities by extending base views or mixing in additional behaviors.

- Better Abstraction: APIView abstracts many complexities of web APIs, allowing developers to focus on application logic rather than boilerplate code.

Conclusion

APIView is an essential tool in the Django REST Framework that elevates the process of building web APIs. By providing a host of features tailored for API

development, it makes the developer's job much simpler and more streamlined.

Create first APIView

Assets and Materials:

- Django Framework (Acquired by executing `pip install django` in the terminal or command prompt)

- Django REST Framework (DRF) (Acquired by executing `pip install djangorestframework` in the terminal or command prompt)

- Atom Editor (Downloaded from atom.io)

- Python (Installed on the development machine)

- Virtual Environment (Created in previous chapters)

- ModHeaders (Installed as an extension in your browser)

Introduction:

APIView, a core part of the Django REST Framework, provides a way to define the logic that gets executed for different types of HTTP methods. Unlike the regular Django views, which handle mostly webpage requests and responses, an APIView is designed to handle API endpoints, returning JSON or XML responses suitable for consumption by other software or front-end frameworks.

Step 1: Setting up the Stage

Before we dive into the APIView creation, ensure you have your Django project set up and the Django REST Framework installed. If not, revisit the previous chapters to make sure your environment is ready.

Step 2: Creating an App for the API

For the sake of clarity, we'll be creating an API for a simple model – let's say, "Messages." Start by creating an app:

“`bash

python manage.py startapp messages_app

“`

Once you've created your app, add it to the `INSTALLED_APPS` list in your project's `settings.py`:

“`python

INSTALLED_APPS = [

    …

    ‘rest_framework’,

    ‘messages_app’,

]

“`

Step 3: Define a Model

In the `models.py` of `messages_app`, define a simple model:

“`python

from django.db import models

class Message(models.Model):

    content = models.TextField()

    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):

        return self.content[:50]

“`

Run migrations:

“`bash
python manage.py makemigrations
python manage.py migrate
“`

Step 4: Create a Serializer

Serializers allow complex data types, like Django models, to be converted to a format that can be easily rendered into JSON. In the `messages_app` directory, create a file named `serializers.py`:

“`python
from rest_framework import serializers
from .models import Message
class MessageSerializer(serializers.ModelSerializer):
    class Meta:
        model = Message
        fields = [‘id’, ‘content’, ‘created_at’]
“`

Step 5: Creating our first APIView

Now, let’s dive into the main topic – the APIView. In the `views.py` file of your `messages_app`, start by importing necessary libraries:

“`python
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Message
from .serializers import MessageSerializer
“`

Next, define your APIView:

“`python

class MessageList(APIView):

    ”””

    List all messages or create a new message.

    ”””

    def get(self, request):

        messages = Message.objects.all()

        serializer = MessageSerializer(messages, many=True)

        return Response(serializer.data)

    def post(self, request):

        serializer = MessageSerializer(data=request.data)

        if serializer.is_valid():

         serializer.save()

         return Response(serializer.data, status=status.HTTP_201_CREATED)

        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

“`

In the above code:

- The `get` method fetches all the messages and returns them.

- The `post` method creates a new message.

Step 6: Configuring the URL

To make your APIView accessible, you need to wire it up with a URL. In the `urls.py` of your `messages_app`:

“`python

from django.urls import path

```
from .views import MessageList

urlpatterns = [

    path('messages/', MessageList.as_view(), name='message-list'),

]
```

Make sure this app's URLs are included in the main project's `urls.py`.

Step 7: Testing with ModHeaders

With everything set up, start your Django server:

```bash
python manage.py runserver
```

Open your browser, navigate to the ModHeaders extension, and set the header "Content-Type" to "application/json". Now, you can visit ` http://127.0.0.1:8000/messages/ ` to see your messages. You can also use tools like Postman or CURL to test the POST method and create new messages.

Conclusion:

Congratulations on creating your first APIView with Django REST Framework! This fundamental building block will be instrumental as you dive deeper into crafting more complex APIs in the subsequent chapters.

Configure view URL

Assets, Resources, and Materials for this Chapter:

1. Django: We'll be using Django, the web framework for building web applications. (How to acquire: Use the command `pip install django==2.2` in your terminal).

2. Django REST Framework: This provides us with tools to build web APIs. (How to acquire: Use the command `pip install djangorestframework==3.9` in your terminal).

3. Text Editor: Atom, as mentioned in previous chapters. (How to acquire: Visit the Atom website at atom.io and download the appropriate version for your operating system).

4. Project Files: Ensure you have your Django project and the associated app set up, as covered in previous chapters.

5. views.py File: This is where we've defined our API views and will be integral in this chapter. (Location: In your Django app directory).

Introduction

One of the primary reasons Django is so popular for web development is because of its built-in URL routing. This chapter will teach you how to configure URLs for your API Views in Django so that users can access the API endpoints you've set up.

Step-by-Step Guide to Configure View URL

1. Import Required Libraries

Before we can set up our URLs, we need to import a few necessities. Open your `urls.py` file in your Django app directory. If you don't have a `urls.py` in your app directory, create one. Now, import the necessary libraries:

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

Here, `path` is a function we use to define the URL pattern, and we're importing our views so we can link

them to these URLs.

2. Define the URL Patterns List

In Django, URLs are defined in a list named `urlpatterns`. This list will contain all the routes for our application. If you have existing routes, we'll simply add our new one. Otherwise, initialize the list:

“`python

urlpatterns = []

“`

3. Add API View to URL Patterns

Recall from the previous chapter where we created our first APIView. We will now link this view to a URL. Add the following line inside the `urlpatterns` list:

“`python

path('api-view/', views.OurApiView.as_view(), name='api-view')

“`

Here:

- `'api-view/'` is the URL endpoint. When users visit this URL, they will access the API view we created.

- `views.OurApiView.as_view()` tells Django to use the `OurApiView` we created in our views. The `as_view()` method is a built-in Django method to convert class-based views to function-based views which are needed for URL routing.

- `name='api-view'` is the name we give this URL pattern. It's useful for reverse URL matching.

4. Include App URLs in Project URLs

If you haven't done so yet, you must include the app's URLs in your project's main `urls.py` (located in the main project folder, not the app folder). Open the project's `urls.py` and make sure you have:

“`python

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('<your_app_name>.urls')),
]
```

Here, `include('<your_app_name>.urls')` essentially imports all the URL patterns you defined in your app's `urls.py`.

Test Your Setup

To verify if everything's set up correctly, run the development server:

```bash
python manage.py runserver
```

Visit ` http://127.0.0.1:8000/api-view/ ` in your web browser. You should see the response from your API view.

Conclusion

Congratulations! You have successfully configured the URL for your API view. This fundamental step ensures that your backend services are accessible via clear and logical web addresses. In the next chapters, we will delve deeper into the power of Django and the Django REST framework, enabling you to create more complex and robust web APIs.

Testing our API View

Assets, Resources, and Materials:

1. Django REST Framework (DRF): Used for building APIs in Django. (Acquire by running the command `pip install djangorestframework`)

2. Postman: A popular tool for testing APIs. (Download and install from [Postman's official website](https://www.postman.com/downloads/))

3. Python's built-in unittest library: Used for writing test cases in Django. (Comes pre-installed with Python)

4. Sample Data: Some sample data to test POST, PUT, PATCH, and DELETE operations.

5. ModHeader Browser Extension: Helps in setting headers for requests. (Install from browser's extension store)

Introduction:

After setting up our APIView, it's imperative to ensure it's working as expected. Testing is not merely a phase in software development, but it's a crucial practice that determines the robustness and reliability of our application. In this chapter, we'll take a deep dive into testing the functionality of our API view.

Step 1: Manual Testing Using Postman

Before we delve into writing automated tests, it's beneficial to do some manual testing using Postman.

1.1. Setting Up Postman

- Launch Postman after installation.

- Create a new request.

- Set the request type. For our first test, we'll use `GET`.

- Enter the URL of our API endpoint.

- If authentication is required, navigate to the `Authorization` tab and provide the necessary

credentials.

1.2. Sending Requests

- For `GET`: Simply click on the `Send` button. Examine the response. It should correspond with the expected data from our API.

- For `POST`: Change the request type to `POST`. Go to the `Body` tab and provide the required data in JSON format. Hit `Send` and check the response.

Remember to test other HTTP methods like `PUT`, `PATCH`, and `DELETE` similarly.

Step 2: Automated Testing Using Django's Unittest

While manual testing is great for a quick check, we need automated tests to ensure our API's robustness.

2.1. Setting Up the Test File

Navigate to the app's directory and create a file named `test_api.py`.

2.2. Writing Test Cases

Here's a simple example of how you'd test the GET method:

“`python

```
from rest_framework.test import APIClient
from rest_framework import status
from django.urls import reverse
class APIViewTestCase(unittest.TestCase):
    def setUp(self):
        self.client = APIClient()
        self.api_url = reverse('name_of_the_view')
    def test_api_get_request(self):
        response = self.client.get(self.api_url)
```

```
        self.assertEqual(response.status_code,
status.HTTP_200_OK)
```

Remember to write similar tests for `POST`, `PUT`, `PATCH`, and `DELETE`.

2.3. Running the Test

Navigate to the project's root directory and run the following command:

```
python manage.py test app_name
```

Replace `app_name` with the name of your app.

Step 3: Using ModHeader for Setting Headers

In some cases, you might need to set headers for your requests. With the ModHeader browser extension, you can easily set headers like token-based authentication.

3.1. Setting Up ModHeader

After installing the extension:

- Open it by clicking its icon in your browser.
- Add a new header by clicking the `+` sign.
- Input the header name and value.

3.2. Using ModHeader with Postman

If your API requires headers like a token for authentication, ensure you set it in Postman during manual testing.

Conclusion:

Testing is a pivotal practice in the API development process. Through tools like Postman and Django's unittest, we can ensure that our API behaves as

expected, providing a solid foundation for further development. Remember always to iterate and refine your tests as you continue developing new features or modifying existing ones.

Create a Serializer

Assets, Resources, and Materials:

- Django (Get it from [Official Django Website](https://www.djangoproject.com/download/))

- Django REST framework (Install using pip: `pip install djangorestframework`)

- Atom or any other code editor of choice (Get Atom from [Atom's Official Site](https://atom.io/))

Introduction:

Serialization is the process of transforming complex data types, such as querysets and model instances, into native data types that can be easily rendered into JSON, XML, or other content types. In Django REST framework, serializers allow complex data types to be converted to easily renderable types, similar to Django's Form and ModelForm classes. In this chapter, we'll focus on creating a serializer using Django REST framework to represent our data in a way that's suitable for rendering into JSON.

Steps to Create a Serializer:

1. Setting up:

   First and foremost, ensure you have the Django REST framework installed. If not, install it using pip:

   ```
   pip install djangorestframework
   ```

2. Creating a Basic Serializer:

Navigate to your Django app directory and create a new file named `serializers.py`. This file will contain all the serializers for your app.

In `serializers.py`, start by importing the required modules:

```python
from rest_framework import serializers
from .models import YourModelName
```

Replace `YourModelName` with the name of the model you wish to serialize.

3. Define the Serializer Class:

Next, you'll define a serializer class for your model:

```python
class YourModelNameSerializer(serializers.Serializer):
    field_name1 = serializers.CharField(max_length=100)
    field_name2 = serializers.DateField()
    # Add more fields as per your model.
```

Here, replace `YourModelNameSerializer` with an appropriate name and define all the fields from your model in the serializer.

4. Using ModelSerializer for Simplicity:

Instead of manually defining each field, you can use the `ModelSerializer` class which will automatically create a serializer with fields that correspond to the model fields. This is similar to how Django's `ModelForm` works:

```python
```

```
class YourModelNameModelSerializer(serializers.ModelSerializer):
    class Meta:
        model = YourModelName
        fields = ['field_name1', 'field_name2', 'field_name3']
        # You can also use fields = '__all__' to include all fields.
```

5. Validating Data with Serializers:

Serializers also allow you to validate data easily. For instance, if you want to ensure that a certain character field does not contain any numbers, you can do:

```python
def validate_field_name1(self, value):
    if any(char.isdigit() for char in value):
        raise serializers.ValidationError("This field should not contain numbers.")
    return value
```

Add this method inside your serializer class.

Testing the Serializer:

It's essential to test your serializer to ensure it's functioning correctly. You can do this using Django's shell:

1. Run the Django shell:

```
python manage.py shell
```

2. Import your model and serializer:

```python
from your_app_name.models import YourModelName
from your_app_name.serializers import YourModelNameSerializer
```

3. Create a model instance and serialize it:

```python
instance = YourModelName.objects.create(field_name1="Test Data", field_name2="2023-10-20")
serializer = YourModelNameSerializer(instance)
print(serializer.data)
```

This will display the serialized data in the console.

Conclusion:

Serializers in the Django REST framework play a pivotal role in transforming complex data into a format that's easy to render and work with on client-side applications. By following the steps outlined in this chapter, you've laid down the groundwork for effectively serializing your data, making it easier to work with APIs.

Practice Exercises:

1. Create serializers for two other models in your application.

2. Test your serializers in the Django shell to ensure they work correctly.

3. Add custom validation to one of the fields in your serializer. Test this validation in the Django shell.

Add POST method to APIView

Assets, Resources, and Materials for this Chapter:

- Django and Django REST Framework: Download and install them using pip (`pip install django djangorestframework`).

- A development environment: We're using Atom in this book, but any IDE or text editor will do.

- A functioning Django project (as set up in previous chapters).

- `serializers.py` file (created in Chapter 29).

Purpose: The POST method allows the client to submit data to the server to be processed as a new entity. In the context of our REST API, it means creating new records in our database.

Introduction

APIView in Django REST Framework provides an easy way to handle HTTP methods like GET, POST, PUT, etc., on a per-method basis, rather than using a single class or function. In this chapter, we'll focus on implementing the POST method to allow clients to create new entities in our system.

---

1. Understand the Flow

Before we dive into the code, it's essential to understand the flow:

1. The client sends a POST request with data.

2. The APIView handles this request.

3. The serializer validates the data.

4. If valid, the data gets saved into the database.

5. A response is returned to the client.

---

2. Update `serializers.py`

Before adding the POST method, ensure you've defined a serializer for the data you'll be accepting. We created a basic serializer in Chapter 29, so we'll use that.

“`python

serializers.py

from rest_framework import serializers

class YourEntitySerializer(serializers.Serializer):

    name = serializers.CharField(max_length=100)

    description = serializers.CharField()

    # Add other fields as required.

“`

---

3. Implementing POST in APIView

Navigate to your views file where your APIView is located. We'll add a post method here.

“`python

views.py

from rest_framework.views import APIView

from rest_framework.response import Response

from rest_framework import status

from .serializers import YourEntitySerializer

class YourEntityApiView(APIView):

    def post(self, request):

        serializer = YourEntitySerializer(data=request.data)

        if serializer.is_valid():

         name = serializer.validated_data.get(‘name’)

```
        # Handle saving data to the database.
        # For now, we'll just return the name.
        return Response({'name': name},
status=status.HTTP_201_CREATED)
      else:
        return Response(
          serializer.errors,
          status=status.HTTP_400_BAD_REQUEST
        )
```

---

4. Handling Database Save

For now, we've only returned the name in the response. In a real-world scenario, you would save this data to the database.

For example, if you're working with a Django model:

```python
from .models import YourEntity
# within the post method, after checking serializer is valid:
entity = YourEntity.objects.create(name=name, description=description)
return Response({'id': entity.id, 'name': entity.name}, status=status.HTTP_201_CREATED)
```

---

5. Updating URLs

Ensure your APIView is hooked up to a URL endpoint to test the POST method. In `urls.py`:

```python
from django.urls import path
```

```
from .views import YourEntityApiView

urlpatterns = [

    path('entity/', YourEntityApiView.as_view(), name='entity-create')

]
```

---

6. Testing POST Request

To test, you can use tools like Postman or CURL:

```bash

curl -X POST -H "Content-Type: application/json" -d '{"name": "Sample Name", "description": "Sample Description"}' http://localhost:8000/entity/

```

You should receive a response with HTTP 201 status and the name you provided.

Conclusion

The POST method is essential for creating new resources in a REST API. With Django REST Framework's APIView and serializers, we can efficiently implement and validate incoming data.

Test POST Function

Assets, Resources, and Materials:

- Postman *(or any other API testing tool like Insomnia)* : An API testing tool used to send requests to our API and view responses. Available for free download at [https://www.postman.com/downloads/] (https://www.postman.com/downloads/) .

- Source Code: From the previous chapters, especially from Chapter 30 where we added the POST method to

our APIView.

Introduction:

In the previous chapter, we added a POST method to our APIView, allowing clients to create new data entries. This chapter focuses on testing that POST function, ensuring it behaves as expected.

Step 1: Setting up Postman:

1.1. Install and Open Postman:

- If you haven't already, download and install Postman from the link provided above.

- Launch Postman once the installation is complete.

1.2. Setup New Request:

- Click on the '+' tab to open a new request.

- Set the request type to "POST" using the dropdown menu.

- Enter your API endpoint URL. For instance, if you're running your Django server locally, it might be something like ` http://127.0.0.1:8000/api/your-endpoint/ `.

Step 2: Constructing the POST Request:

2.1. Headers:

- Set the "Content-Type" to "application/json". This tells our API that we're sending data in JSON format.

2.2. Body:

- Click on the "Body" tab below the URL field.

- Choose "raw" input, and ensure "JSON (application/json)" is selected on the right dropdown.

- Input your data in JSON format. For instance, if you're testing a user creation API, it might look something like:

"`json

```
{
    “username”: “testuser”,
    “email”: “testuser@email.com”,
    “password”: “securepassword123”
}
“`
```

Step 3: Sending the POST Request:

3.1. Click the blue “Send” button. Postman will send the POST request to your API.

3.2. Analyze the Response:

- Once the request completes, Postman will display the response below. Typically, for a successful POST request, you’d expect a 201 status code indicating the resource was successfully created.

- Examine the response body for the returned data. It should ideally show the created object with any system-generated fields, such as an ID.

Step 4: Handling Errors:

4.1. If the POST request returns an error, analyze the response body carefully. Django and the Django REST Framework typically return descriptive error messages that can guide you in resolving the issue.

4.2. Common errors include:

- 400 Bad Request: Typically signifies that the data you sent is not valid. Check if all required fields are present and have valid values.

- 403 Forbidden: Indicates a permissions issue. Ensure that your API is set up to allow POST requests from your testing source.

- 500 Internal Server Error: A generic error. Check your Django server logs for more specific details.

Step 5: Additional Tests:

5.1. To thoroughly test the POST function:

- Try sending incomplete or invalid data to check if your API correctly validates incoming data and returns appropriate error messages.

- If your API uses authentication, test with both authenticated and unauthenticated requests to ensure security measures are functioning.

- Test the idempotence. If you send the same POST request multiple times, does it create multiple entries (if it's supposed to) or does it reject duplicates?

Conclusion:

Testing is a crucial step in the development process, ensuring that our APIs function as intended and are ready for production. By thoroughly testing our POST function, we can be confident in its behavior and its readiness to handle real-world data from users or other services.

Add PUT, PATCH and DELETE methods

Assets, Resources, and Materials required for this chapter:

- Python (Get it from [https://www.python.org/downloads/] (https://www.python.org/downloads/)): The programming language in which Django is written.

- Django (Install via pip: `pip install django==2.2`): Our main web framework.

- Django REST Framework (DRF) (Install via pip: `pip install djangorestframework==3.9`): This provides tools to create Web APIs.

- Postman (Get it from [https://www.postman.com/downloads/] (https://www.postman.com/downloads/)): An API client used for testing.

- Your existing Django project: You should have the project set up from the previous chapters.

Introduction

By now, you've learned about APIViews in Django Rest Framework (DRF), and you've already created a GET endpoint to retrieve information. But most APIs aren't just about retrieving data; they also allow clients to modify data. In this chapter, we're going to add three methods to our APIView: PUT (for updating objects entirely), PATCH (for partial updates), and DELETE (for removing objects).

Step 1: Understanding HTTP Methods

Before diving into the coding, let's understand what each method is meant to do:

- PUT: This method is used to update the entire object. If any field is missing in the request, it will be assumed that the missing fields should be set to their default values.

- PATCH: This method allows for partial updates. So, if you only want to change one specific part of an object but leave the rest unchanged, you'd use PATCH.

- DELETE: This method is straightforward — it's used to delete objects.

Step 2: Modify your APIView

Now, let's add the three methods to our existing APIView. Navigate to your `views.py` where your

APIView resides.

“`python

```
from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework import status
# (Assuming you already have a serializer and model imported)
class YourModelNameApiView(APIView):
    … # Your previously written methods here
    def put(self, request, pk=None):
        """Update an object entirely"""
        obj = self.get_object(pk)
        serializer = YourModelNameSerializer(obj, data=request.data)
        if serializer.is_valid():
         serializer.save()
         return Response(serializer.data, status=status.HTTP_200_OK)
        else:
         return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    def patch(self, request, pk=None):
        """Update an object partially"""
        obj = self.get_object(pk)
        serializer = YourModelNameSerializer(obj, data=request.data, partial=True)
        if serializer.is_valid():
         serializer.save()
         return Response(serializer.data, status=status.HTTP_200_OK)
```

```
        else:
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
    def delete(self, request, pk=None):
        """Delete an object"""
        obj = self.get_object(pk)
        obj.delete()
        return
Response(status=status.HTTP_204_NO_CONTENT)
```

Step 3: Testing the Methods

Once you've added the methods, it's essential to test them to ensure they work as expected.

1. Testing PUT:
   - Launch Postman.
   - Set the request type to PUT.
   - Provide the endpoint URL.
   - In the body, set the data you want to update.
   - Send the request.
   - The response should reflect the updated data.
2. Testing PATCH:
   - Similarly, in Postman, set the request type to PATCH.
   - Provide the endpoint URL and only the specific data you want to update.
   - Send the request.
   - The response should reflect the partial update.
3. Testing DELETE:
   - In Postman, set the request type to DELETE.

- Provide the endpoint URL for the object you wish to delete.

- Send the request.

- You should get a 204 No Content response, indicating a successful deletion.

Conclusion

The PUT, PATCH, and DELETE methods are essential for a functional API, allowing users to update and remove data as needed. By now, your APIView should be capable of handling all the CRUD operations, making it a complete and functional endpoint.

In the next chapter, we'll test these methods more extensively to ensure they handle all scenarios and edge cases.

That concludes our chapter on adding PUT, PATCH, and DELETE methods in an APIView. Please remember to always test thoroughly after implementing new features to ensure everything works as intended.

Test the PUT, PATCH and DELETE methods

Assets, Resources and Materials:

- Postman (To send PUT, PATCH, and DELETE requests to our API. Download and install it from [Postman's official website](https://www.postman.com/downloads/). Purpose: Postman is an API testing tool which helps to send requests to an API and receive responses.)

- Your Django Development Server (Must be running. We set it up in previous chapters.)

- API URL (The endpoint of our previously created APIView.)

Introduction:

In our previous chapters, we set up our APIView and added PUT, PATCH, and DELETE methods to it. In this chapter, we will test these methods to ensure they are working correctly. Testing is a crucial aspect of web development to ensure that your application behaves as expected.

---

1. Setting Up Postman:

Before we begin, ensure you have Postman installed. Once installed:

1. Open Postman.

2. Create a new request.

3. From the dropdown on the left side, select the type of request. We will start with PUT.

---

2. Testing the PUT Method:

The PUT method is used for updating resources. Let's test it:

1. In Postman, enter the API URL of the resource you want to update.

2. Select the PUT request method.

3. Go to the 'Body' tab.

4. Choose 'raw' and 'JSON' format.

5. Add the data you want to update in JSON format.

6. Click 'Send'.

You should receive a response indicating the resource has been updated. If not, check for error messages to debug the issue.

Example:

Let's say you have an API endpoint at ` http://localhost:8000/api/profile/1/ `, representing the

profile with ID 1.

If you want to update the name for this profile, your JSON data in the body might look like:

“`json

{

“name”: “Updated Name”

}

“`

---

3. Testing the PATCH Method:

The PATCH method is used for partial updates, unlike PUT which updates the resource completely. The steps to test PATCH are similar to PUT:

1. In Postman, enter the API URL of the resource you want to partially update.

2. Select the PATCH request method.

3. Go to the ‘Body’ tab.

4. Choose ‘raw’ and ‘JSON’ format.

5. Add the data you want to update in JSON format.

6. Click ‘Send’.

Again, ensure you receive a response indicating the partial update was successful.

Example:

Using the same endpoint as before, if you only want to update the email for the profile and leave other attributes unchanged, your JSON data might be:

“`json

{

“email”: “updated.email@example.com”

}

“`

---

4. Testing the DELETE Method:

DELETE is straightforward. It’s used to remove a resource:

1. In Postman, enter the API URL of the resource you want to delete.

2. Select the DELETE request method.

3. Click ‘Send’.

After sending the DELETE request, you should get a response indicating the resource has been deleted.

Note: Use the DELETE method with caution during testing, as it will remove the data. You may need to re-seed your database or recreate resources if you want to continue testing other functionalities.

Conclusion:

By now, you should have successfully tested the PUT, PATCH, and DELETE methods of your APIView. It’s always recommended to perform rigorous testing, especially with different scenarios and edge cases, to ensure the robustness of your API.

Homework:

- Try using different data inputs and observe the API responses.

- See how your API responds if you try to send a PATCH or PUT request without any body data or with incorrect data types.

Section 9:

Introduction to Viewsets

What is a Viewset?

Assets, Resources, and Materials:

- Django REST Framework (DRF): You will need the Django REST Framework installed. It's a powerful and flexible toolkit for building Web APIs in Django. (To install: `pip install djangorestframework`)

- Django Project: Ensure you have a Django project set up with the DRF integrated. If you have followed the previous chapters, you should already have this ready.

- Python Environment: You should be working within a Python virtual environment to avoid potential package conflicts. (Use `virtualenv` or the built-in `venv` module)

- Text Editor: We will be using Atom, as mentioned earlier, but any text editor or IDE of your choice will work.

- Documentation: The [official DRF documentation](https://www.django-rest-framework.org/) is an invaluable resource. Bookmark it for easy access.

Introduction

In the realm of Web APIs, especially when working with Django REST Framework (DRF), you might often hear about "views" and "viewsets". While you're familiar with the concept of views in Django, the term "viewset" might be new to you. This chapter is dedicated to unraveling the mystery behind viewsets, making them an accessible and integral tool in your DRF toolkit.

What is a Viewset?

In the simplest terms, a Viewset is a high-level abstraction provided by DRF that combines the logic for handling HTTP methods (such as GET, POST, PUT,

DELETE) into a single class. It's a layer on top of Django's views, designed specifically for working with serialized data and querysets.

Think of Viewsets as a bridge between your models (database) and your serializers (representation). They decide how your data gets processed and what gets sent as a response or expected as input.

Key Features of Viewsets:

1. CRUD Operations: Viewsets inherently understand the concept of CRUD operations (Create, Read, Update, Delete). When you define a viewset, you're inherently saying, "I want to create a set of views that handle these CRUD operations for a particular model and serializer."

2. Fewer Lines of Code: One of the significant advantages of using viewsets is the reduction in the amount of code. With traditional views, you might need separate classes or methods for list views and detail views. Viewsets encapsulate this behavior under one roof.

3. Routers: Another power of viewsets lies in their seamless integration with DRF's routers. This allows for automatic URL pattern creation for your API endpoints.

4. Permissions and Authentication: Just like with APIViews, you can easily integrate authentication and permissions with viewsets, ensuring that your data is secure and only accessible to authorized users.

5. Customization: While viewsets provide a lot of out-of-the-box functionality, they're also highly customizable. You can override methods, add extra actions, or integrate with third-party packages.

Difference Between APIViews and Viewsets

To further clarify the concept of viewsets, let's differentiate them from APIViews:

- APIView: This is a DRF abstraction that handles HTTP methods as separate class methods. For example, for a `GET` request, you would implement the `get()` method. It provides more control over the logic of responses and requests.

- Viewset: This is higher-level than APIViews and abstracts the logic of handling CRUD operations based on your model's queryset. It's best suited for standard database operations. For any non-standard behavior, you might still revert to APIViews or further customize the viewset.

Conclusion

In essence, viewsets in the Django REST Framework offer a powerful and streamlined way to create API endpoints for your serialized data. They reduce boilerplate code and offer easy integration with routers, making the process of building a robust Web API smoother and more intuitive.

In the upcoming chapters, we'll dive deeper into how to create and utilize viewsets effectively, ensuring you have a holistic understanding of this critical DRF component.

Create a simple Viewset

Assets, Resources, and Materials:

- Python: This is the primary language we will use for our backend. (Can be downloaded and installed from [python.org](https://www.python.org/downloads/)).

- Django: A high-level Python web framework that we'll use to create our application. (Install via pip with `pip install django`).

- Django REST Framework (DRF): An extension to Django, it makes building RESTful APIs simpler. (Install via pip with `pip install djangorestframework`).

- Code Editor (e.g., Atom): This is where you'll be writing and editing your code. (atom.io).

- Terminal or Command Line: To run our server, apply migrations, and more.

- Web Browser: To view and test our API. (You can use Chrome, Firefox, Safari, or any other browser of your choice).

Introduction:

Viewsets are a key component in Django REST Framework. They allow developers to rapidly build CRUD operations without having to define methods for each specific HTTP verb. Viewsets combine the logic for handling HTTP methods (GET, POST, PUT, DELETE) into a single class for a particular model.

For this chapter, we'll be building a simple Viewset for a hypothetical `Book` model, which will be a basic representation of books with fields like title, author, and publication_date. Let's get started.

Step 1: Define the Book Model

Before we can create a Viewset, we need a model.

In your models.py:

“`python

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=100)
    publication_date = models.DateField()
    def __str__(self):
        return self.title
```

“`

Step 2: Create Serializers

Serializers allow complex data types to be converted to Python data types that can be rendered easily into JSON.

In your serializers.py:

“`python

from rest_framework import serializers

from .models import Book

class BookSerializer(serializers.ModelSerializer):

    class Meta:

        model = Book

        fields = [‘id’, ‘title’, ‘author’, ‘publication_date’]

“`

Step 3: Create a Simple Viewset

With our model and serializer ready, it’s time to create a Viewset.

In your views.py:

“`python

from rest_framework import viewsets

from .models import Book

from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer_class = BookSerializer

“`

Here, `viewsets.ModelViewSet` provides default `list()`, `create()`, `retrieve()`, `update()`, and `destroy()` actions. The `queryset` and `serializer_class` attributes tell the

Viewset which database records to work with and how to convert between database records and JSON.

Step 4: Wire up the Viewset to URLs

In your urls.py:

“`python

from django.urls import path, include

from rest_framework.routers import DefaultRouter

from .views import BookViewSet

router = DefaultRouter()

router.register(r’books’, BookViewSet)

urlpatterns = [

    path(”, include(router.urls)),

]

“`

The DRF router automatically generates URLs for our Viewset actions.

Conclusion:

With these steps, you’ve set up a basic Viewset for the `Book` model. When you run your Django server and navigate to the related endpoint (e.g., `/books/`), you can now perform CRUD operations on the `Book` model via the API.

As this is an introduction, we’ve used the simplest version of a Viewset. As you continue in your learning, you’ll discover more functionalities and nuances of Viewsets and other components of Django REST Framework.

Add URL Router

Assets, Resources, and Materials for this Chapter:

- Django: A Python web framework used to build web applications.

  (How to acquire: You can install Django using pip by running `pip install django`)

- Django REST Framework (DRF): An extension for Django used for building Web APIs.

  (How to acquire: Install it via pip with `pip install djangorestframework`)

- Django's Default Router: A part of Django REST Framework which helps in automatically determining URL patterns for ViewSets.

  (Included with DRF installation)

Introduction

When working with Viewsets in Django REST Framework (DRF), the URL router plays a crucial role in automatically generating URL patterns. Instead of manually defining each URL pattern, as you might do with APIViews, Viewsets combined with DRF's router help streamline this process, making it both efficient and elegant.

Understanding the DRF's Router

Django REST Framework includes a set of routers that assist in automatically creating the appropriate URLs for your viewsets. The most commonly used router is the `DefaultRouter`.

The `DefaultRouter` class will automatically create the URL patterns for you and provide a simple interface for registering your viewsets.

Step-by-Step Guide to Adding URL Router

1. Import Necessary Libraries

At the top of your `urls.py` (where you define your URL patterns), you'll need to import the necessary modules:

“`python

from django.urls import path, include

from rest_framework.routers import DefaultRouter

“`

2. Create an Instance of the DefaultRouter

Next, you'll create an instance of the `DefaultRouter`:

“`python

router = DefaultRouter()

“`

3. Register the Viewset with the Router

For this example, let's assume you have a Viewset named `UserProfileViewSet` in a file called `views.py`.

First, import the viewset at the top of your `urls.py`:

“`python

from .views import UserProfileViewSet

“`

Now, you can register this viewset with the router:

“`python

router.register(r'profiles', UserProfileViewSet)

“`

The first argument, `r'profiles'`, specifies the prefix for our URL patterns. This means our API endpoint for user profiles would look something like `http://yourdomain.com/profiles/`.

4. Add Router URLs to Django's URL Patterns

Finally, you need to include the router's URL patterns into Django's URL patterns:

“`python

```
urlpatterns = [
    path('', include(router.urls)),
]
```

This tells Django to include all of the URLs associated with the router.

Final Result

With everything put together, your `urls.py` should look something like this:

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import UserProfileViewSet
router = DefaultRouter()
router.register(r'profiles', UserProfileViewSet)
urlpatterns = [
    path('', include(router.urls)),
]
```

With this setup, your UserProfileViewSet will have a set of standardized URLs. For example:

- List all profiles: `GET /profiles/`
- Retrieve a specific profile: `GET /profiles/{id}/`
- Update a profile: `PUT /profiles/{id}/`
- Partially update a profile: `PATCH /profiles/{id}/`
- Delete a profile: `DELETE /profiles/{id}/`

Conclusion

In this chapter, you learned the importance of DRF's router, specifically the `DefaultRouter`, and how it greatly simplifies URL management for ViewSets. With this setup, not only does it make the code cleaner, but it also ensures a consistent URL pattern across your web API, making it more predictable and user-friendly.

Testing our Viewset

Assets, Resources, and Materials:

- Python: The primary language we'll use for backend scripting. (Available at [python.org](https://www.python.org/))

- Django: A high-level web framework for building web applications. (Install using pip: `pip install django==2.2`)

- Django REST Framework (DRF): A powerful and flexible toolkit for building Web APIs. (Install using pip: `pip install djangorestframework==3.9`)

- Postman: A popular tool for testing APIs. (Available at [postman.com](https://www.postman.com/downloads/))

- unittest: Python's built-in library for unit testing. (Included with Python)

Introduction

In our previous chapters, we've walked through the creation and setup of a Viewset. Now, we must ensure that the functions we've written are not only working correctly but also providing the right responses. Testing is essential to maintain the reliability and stability of your code as you add or modify features. In this chapter, we'll take a close look at how you can test your Viewset in Django using both manual and automated methods.

Manual Testing using Postman

Before we dive into automated testing, let's first use Postman to manually test our Viewset endpoints.

Steps:

1. Install and Setup Postman: Download and install Postman from their official website. Once installed, open it up.

2. Setup the Request: Choose the type of request (GET, POST, PUT, PATCH, DELETE) based on the Viewset function you want to test. Input your API endpoint URL.

3. Add Data (if required): For POST, PUT, and PATCH requests, go to the 'Body' tab and input any necessary data in JSON format.

4. Send Request and Examine the Response: Click the 'Send' button. Once the request is processed, you'll see the server's response below. Ensure the response data and status code match your expectations.

Automated Testing using `unittest`

While manual testing is good for initial checks, we can't rely on it for large applications. It's time-consuming, and there's always a chance you might miss out on testing a particular scenario. Automated tests, once written, can be run any number of times, ensuring consistent testing every time.

Setting Up Your Test Environment:

1. In your Django project, create a file named `tests.py` within your app folder.

2. In `tests.py`, import the necessary libraries:

"`python

from django.test import TestCase

from rest_framework.test import APIClient

from .models import YourModel

“`

Writing Your First Test:

Let’s test if our Viewset returns a list of items when we make a GET request:

“`python

class ViewsetTestCase(TestCase):

    def setUp(self):

        self.client = APIClient()

        YourModel.objects.create(name=“TestItem1”, description=“TestDescription1”)

        YourModel.objects.create(name=“TestItem2”, description=“TestDescription2”)

    def test_get_items(self):

        response = self.client.get(‘/your_endpoint_url/’)

        self.assertEqual(response.status_code, 200)

        self.assertEqual(len(response.data), 2)

“`

1. Setup: Here, we initialize the APIClient (which will mimic our API calls) and set up some sample data using our model.

2. Test: We’re sending a GET request to our endpoint. We then check if the response status code is 200 (OK) and if we receive two items in our response.

Running Your Test:

To run your test, navigate to your project’s root directory in the terminal and run:

“`bash

python manage.py test your_app_name

“`

Ensure all tests pass. If any fail, they'll provide error messages which you can use to debug and fix the respective parts of your code.

Conclusion

Testing ensures that your Viewsets, and by extension, your APIs, function as intended. Manual testing provides a quick way to check the flow, while automated testing ensures consistent reliability. As you add more features or make changes to your API, ensure that you update and run your tests, guaranteeing stability and reliability for your users.

Add create, retrieve, update, partial_update and destroy functions

Assets, Resources, and Materials:

- Django: A high-level web framework for building web applications. You can download and install Django by running `pip install django==2.2`.

- Django REST framework: A flexible toolkit to build Web APIs. Install using `pip install djangorestframework==3.9`.

- Postman: A popular tool for testing APIs. You can download it from the [Postman official website](https://www.postman.com/downloads/).

- Text Editor (Atom): Download and install Atom from [here](https://atom.io/).

Introduction:

Viewsets are a high-level component in Django REST framework that combines the logic for handling HTTP methods (GET, POST, PUT, etc.) into a single class. In this chapter, we'll implement the core functions (create, retrieve, update, partial_update, and destroy) for our Viewset. These functions will allow our API to handle different types of requests, including creating, retrieving, updating, and deleting objects.

---

1. The Basics of Viewset Functions:

Before diving into the implementation, it's vital to understand the purpose of each function:

- create: Handles HTTP POST requests and is used to create a new object.

- retrieve: Handles HTTP GET requests for a single object. Returns a specific object by its ID.

- update: Handles HTTP PUT requests. Updates an object entirely.

- partial_update: Handles HTTP PATCH requests. Updates specific fields of an object.

- destroy: Handles HTTP DELETE requests. Deletes an object.

---

2. Setting up the Viewset:

In your `views.py`, ensure you have the following imports:

"`python

from rest_framework import viewsets

from .models import YourModel

from .serializers import YourModelSerializer

"`

Replace `YourModel` with the name of your model and `YourModelSerializer` with the serializer for that model.

---

3. Implementing the Functions:

a) Create Function:

“`python

```
def create(self, request):
    serializer = YourModelSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data,
status=status.HTTP_201_CREATED)
    return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

“`

This function will handle the POST request, validate the data using our serializer, and save it to the database.

b) Retrieve Function:

“`python

```
def retrieve(self, request, pk=None):
    queryset = YourModel.objects.all()
    item = get_object_or_404(queryset, pk=pk)
    serializer = YourModelSerializer(item)
    return Response(serializer.data)
```

“`

Here, we’re fetching a specific item using its primary key (ID) and returning its serialized data.

c) Update Function:

“`python

```
def update(self, request, pk=None):
    queryset = YourModel.objects.all()
    item = get_object_or_404(queryset, pk=pk)
```

```
    serializer = YourModelSerializer(item, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

This function will entirely update an object if the provided data is valid.

d) Partial Update Function:

```python
def partial_update(self, request, pk=None):
    queryset = YourModel.objects.all()
    item = get_object_or_404(queryset, pk=pk)
    serializer = YourModelSerializer(item, data=request.data, partial=True)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

This function will update specific fields, given that the request data only contains the fields that need updating.

e) Destroy Function:

```python
def destroy(self, request, pk=None):
    queryset = YourModel.objects.all()
    item = get_object_or_404(queryset, pk=pk)
```

```
    item.delete()
    return
Response(status=status.HTTP_204_NO_CONTENT)
```

It will delete a specific item identified by its primary key.

---

4. Testing the Functions with Postman:

Once you've implemented these functions, you can use Postman to send various HTTP requests to your API. Ensure that you have set up your URLs correctly to test each of the above functions.

Conclusion:

With these functions in place, our Viewset now has the capability to handle various HTTP requests, enhancing the functionality of our REST API. These core functions are the foundation for most CRUD operations in web applications, and mastering them will put you a step ahead in backend development.

Test Viewset

Assets, Resources, and Materials for this chapter:

- Python: The primary language used in this book. If not installed, you can download it from [python.org](https://www.python.org/).

- Django: The web framework used to build our application. If not installed, you can download it using pip: `pip install django`.

- Django REST Framework (DRF): Provides us the tools to create RESTful services. Download using pip: `pip install djangorestframework`.

- Postman: A tool used for testing API endpoints. You can download it from [Postman's official website](

https://www.postman.com/downloads/).

- ViewSet and Router: Covered in previous chapters.

Introduction

As you're delving deeper into Django REST Framework (DRF), you'll encounter an essential component called "Viewsets". They provide a high level of abstraction for your views, making it easier to create, read, update, and delete objects in your database. But with this abstraction comes the responsibility to ensure that your endpoints are functioning as expected. Testing, as a practice, ensures that the endpoints you expose to the outside world are reliable and consistent. In this chapter, we'll look at how to test the Viewset that you've set up.

---

1. Understanding What to Test

Before jumping straight into writing test cases, let's identify the core functionalities of our Viewset:

- List: Can it retrieve and list all objects?

- Retrieve: Can it fetch a specific object by its ID?

- Create: Can it successfully create a new object?

- Update: Can it modify an existing object?

- Partial Update: Can it modify parts of an existing object?

- Delete: Can it delete an object?

Each of these functionalities represents a potential test case.

---

2. Setting up for Testing

Before you begin testing, ensure you have a separate testing database. Django takes care of this for you, so whenever you run tests, it creates a separate database to ensure your main database remains unchanged.

---

3. Writing the Test Cases

Here's a simplified example for testing the `list` functionality of our Viewset:

“`python

from rest_framework.test import APITestCase

from rest_framework import status

from .models import YourModel

class ViewsetTestCase(APITestCase):

    def setUp(self):

        # This creates a sample object that we can use to test our list functionality.

        YourModel.objects.create(name=“Sample”, description=“Sample description”)

    def test_list_objects(self):

        response = self.client.get(‘/path-to-your-viewset/’)

        self.assertEqual(response.status_code, status.HTTP_200_OK)

        self.assertEqual(len(response.data), 1)

“`

Here, we're making a `GET` request to our Viewset, then asserting two things:

1. The status code of our response should be 200 (OK).

2. The length of our response data should be 1, since we created one object in our `setUp` method.

---

4. Testing the Other Functionalities

Follow a similar approach to test `retrieve`, `create`, `update`, `partial update`, and `delete`. Remember to adjust your HTTP method (GET, POST, PUT, PATCH, DELETE) depending on the action you're testing. Use

Postman to help you understand the kind of responses you should expect.

---

5. Running the Tests

Navigate to the root directory of your Django project in the terminal. Run the tests with:

“`bash

python manage.py test

“`

Django will search for all the files that start with the word `test` and execute them. Ensure that all your tests pass. If any fail, the console will provide a descriptive error message to help you identify and fix the problem.

Testing your Viewset ensures that your API behaves as expected. It’s a crucial step in building a robust and reliable application. Always make it a practice to write tests whenever you introduce new features or modify existing ones. This habit will save you countless hours debugging in the future and will give you the confidence that your application works under all scenarios.

Section 10:

Create Profiles API

Plan our Profiles API

Assets, Resources, and Materials:

- User Stories or Use Cases: (These are hypothetical scenarios that detail how a user interacts with a feature or system. They can be created using tools like JIRA,

Trello, or even just paper and pen. Purpose: To help understand user requirements.)

- UML Diagram Tool: (Unified Modeling Language tools like Lucidchart, Draw.io, or even paper and pencil. Purpose: For creating entity relationship diagrams to map out our data structures.)

- API Design Tool: (Tools like Swagger or Postman. Purpose: To visualize the API endpoints and methods.)

Introduction:

Planning an API is a critical step in its development lifecycle. Properly planning ensures that the development process is streamlined and reduces the chance of missing critical features or introducing bugs. For our Profiles API, our main objective is to provide functionality to handle user profiles – including creating, reading, updating, and deleting them.

---

1. User Stories:

Before we begin coding, it's essential to understand what the end user expects from our API. User stories are an effective way to capture these requirements. Here are a few user stories relevant to our Profiles API:

- As a user, I want to create a profile with my name, email, and profile picture.

- As a user, I want to view my profile.

- As a user, I want to update my profile information.

- As a user, I want to delete my profile.

- As a user, I want to search for other users by their names or email addresses.

---

2. Define the Data Model:

With our user stories in place, we need to design the data model for our Profiles API. Based on the

requirements, our Profile model might look like this:

- `User_ID`: Unique identifier for the user.
- `Name`: Full name of the user.
- `Email`: Email address of the user.
- `Profile_Picture`: URL of the profile picture.
- `Created_At`: Date and time when the profile was created.
- `Updated_At`: Date and time when the profile was last updated.

---

3. Designing the API Endpoints:

Using our user stories and data model, we can now plan the endpoints of our Profiles API:

- Create a Profile:
  - Endpoint: `POST /profiles/`
  - Payload: `{ “name”: “John Doe”, “email”: “john@example.com”, “profile_picture”: “url_to_picture” }`
- Get a Profile:
  - Endpoint: `GET /profiles/<User_ID>/`
- Update a Profile:
  - Endpoint: `PUT /profiles/<User_ID>/`
  - Payload: `{ “name”: “John A. Doe”, “email”: “john.a@example.com”, “profile_picture”: “new_url_to_picture” }`
- Delete a Profile:
  - Endpoint: `DELETE /profiles/<User_ID>/`
- Search Profiles:
  - Endpoint: `GET /profiles/?search=<query>`
  - This will search for profiles by name or email.

---

4. API Response and Status Codes:

To ensure our API communicates effectively with the frontend or other services, we need to use appropriate status codes and response messages. For example:

- `200 OK`: Successful GET request.

- `201 Created`: Profile successfully created.

- `204 No Content`: Successful DELETE request.

- `400 Bad Request`: If the request payload is incorrect.

- `404 Not Found`: If a profile with a given ID is not found.

---

5. Security Considerations:

When planning our API, it's crucial to keep security in mind:

- Authentication: Ensure that only authenticated users can create, update, or delete profiles.

- Authorization: A user should only be able to update or delete their own profile, not someone else's.

- Data Validation: Validate all incoming data to prevent malicious input.

- Rate Limiting: Implement rate limiting to prevent abuse.

Conclusion:

By planning our Profiles API in advance, we've laid a clear roadmap for its development. The next steps will involve actual development, where we'll implement our data models, API endpoints, and security measures based on this plan. Remember, a well-planned API often translates to a well-implemented one!

Create user profile serializer

Assets and Resources:

- Django REST Framework (DRF): A powerful and flexible toolkit for building Web APIs. You can install it using pip: `pip install djangorestframework`.

- Django: Ensure you have Django installed. If not, use pip: `pip install Django`.

- Python: The core language in which Django and DRF are written.

- User Model: We would have created this in Chapter 18. The user model serves as the structure for storing user information in the database.

Introduction

Serializers in Django REST Framework (DRF) allow complex data types like Django models to be converted to Python data types. In the context of APIs, these Python data types can then be easily rendered into JSON or XML for client consumption. For our project, the primary use of serializers will be to convert our User Profile instances into JSON format.

In this chapter, we'll focus on creating a serializer for our user profile, which will play a crucial role in the data exchange between the client and the server.

Step 1: Creating the Serializer File

1. Within your Django app folder (where models.py, views.py, etc. are located), create a new file called `serializers.py`.

Step 2: Set up the Serializer

In `serializers.py`:

“`python

from rest_framework import serializers

from .models import UserProfile

class UserProfileSerializer(serializers.ModelSerializer):

```
    class Meta:
        model = UserProfile
        fields = ('id', 'email', 'name', 'password')
        extra_kwargs = {
         'password': {
            'write_only': True,
            'style': {'input_type': 'password'}
         }
        }
```

Let's breakdown what we did:

- Import Necessary Modules: We imported the `serializers` from DRF and our `UserProfile` model.

- UserProfileSerializer: This is our new serializer class which inherits from `serializers.ModelSerializer`.

    - Within the nested `Meta` class:

        - `model = UserProfile`: This ties our serializer to the `UserProfile` model.

        - `fields`: Specifies which fields from the model we want to be included in the serialized representation.

        - `extra_kwargs`: Provides additional options for our serializer. In this case, we're specifying that the password field should be write-only (for security) and its input type should be of type password (so it will be hidden in forms).

Step 3: Serializer Methods (optional but recommended)

To enhance the functionality and security of our API, we can add custom methods to our serializer. For instance, let's ensure that when a user profile is created using the serializer, the password is saved in a hashed format:

```python
```

```
    def create(self, validated_data):
        """Create and return a new user, with encrypted password"""
        user = UserProfile.objects.create_user(
         email=validated_data['email'],
         name=validated_data['name'],
         password=validated_data['password']
        )
        return user
```

With this method, when a new user is created using this serializer, the password is stored securely.

Conclusion

Serializers are an essential part of Django REST Framework applications, allowing us to easily validate and transform data between Python objects and JSON representations. With our `UserProfileSerializer` in place, we're now ready to handle user profile data in our API views and viewsets in subsequent chapters. Remember, serializers not only ensure our data is correctly formatted but also play a pivotal role in the security of our data exchange.

Create profiles ViewSet

Assets, Resources, and Materials:

- Django REST Framework (DRF): To install, use the pip package manager by running `pip install djangorestframework`. We'll use DRF to create our ViewSets.

- Django Project & App: Should already be set up from previous chapters.

- User Profile Model: Created in Chapter 41. This will be the core model we'll be interacting with in our ViewSet.

- Text Editor (e.g., Atom): For writing our code. Any text editor or IDE that you're comfortable with can be used.

- Command-Line Terminal: To run server commands and migrations.

Introduction:

In this chapter, we'll dive deep into creating a ViewSet for our user profiles. ViewSets, courtesy of the Django REST Framework, allow developers to rapidly build CRUD (Create, Read, Update, Delete) operations for their models. By the end of this chapter, you'll have an operational ViewSet for the `UserProfile` model that you created in Chapter 41.

---

1. What are ViewSets in Django REST Framework?

In the Django REST Framework, ViewSets are a high-level abstraction that combine the logic for handling HTTP methods (like GET, POST, PUT, etc.) into classes. Instead of writing individual views for each possible operation (list all profiles, view a single profile, update a profile, etc.), a single ViewSet can replace multiple views by providing these operations out of the box.

---

2. Setting Up The UserProfileViewSet

Navigate to your app's `views.py` file, and let's start setting up the ViewSet.

"`python

```
from rest_framework import viewsets
from .models import UserProfile
from .serializers import UserProfileSerializer
class UserProfileViewSet(viewsets.ModelViewSet):
    queryset = UserProfile.objects.all()
```

```
    serializer_class = UserProfileSerializer
```

Here's a breakdown:

- We import the necessary modules: `viewsets` from DRF, our `UserProfile` model, and the `UserProfileSerializer` we created in the previous chapter.

- We then create a new class called `UserProfileViewSet` that inherits from `viewsets.ModelViewSet`.

- Within this class, we define two class-level variables:

  - `queryset`: This is where we define which records we want to make available. In this case, we want to make all user profiles available.

  - `serializer_class`: This tells the ViewSet to use our `UserProfileSerializer` to serialize the database records into a format that can be rendered into a response.

---

3. Registering the UserProfileViewSet with our URL Configuration

For our ViewSet to be accessible via the web, we need to register it with our URL configuration. First, we need to set up a router. Open your `urls.py`:

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from . import views
router = DefaultRouter()
router.register(r'profiles', views.UserProfileViewSet)
urlpatterns = [
    path('', include(router.urls)),
]
```

“`

Here’s what we did:

- We import necessary modules.

- Create a `DefaultRouter` instance.

- Register our `UserProfileViewSet` with this router. The first argument `r’profiles’` determines the URL prefix for our API endpoint.

- Finally, we include the router’s URLs in our app’s URL patterns.

---

4. Test the UserProfileViewSet

Run your development server:

“`

python manage.py runserver

“`

Visit ` http://127.0.0.1:8000/profiles/ ` in your web browser. You should see a list of all user profiles (if any exist). You can also use tools like Postman or your browser to test POST, PUT, and DELETE requests.

Conclusion:

Congratulations! You’ve now created a fully operational ViewSet for your `UserProfile` model. This ViewSet provides endpoints for listing all profiles, viewing a single profile, updating, creating, and deleting profiles.

Next Steps:

In the next chapter, we’ll register our `UserProfileViewSet` with the URL router to make our API accessible via the web. We’ll also cover how to handle different HTTP methods to interact with our API, such as GET to retrieve data and POST to add new profiles.

Register profile Viewset with the URL router

Assets, Resources, and Materials:

- Django REST Framework: Provides tools to help build a Web API. ([Acquire from](https://www.django-rest-framework.org/))

- Python: The core programming language used in this book. ([Acquire from](https://www.python.org/downloads/))

- Django: The web framework for perfectionists with deadlines. ([Acquire from](https://www.djangoproject.com/download/))

- Code Editor (e.g., Atom): For writing and viewing code. ([Acquire from](https://atom.io/))

- Your previous code: From prior chapters to ensure continuity.

Introduction:

After creating our profile Viewset, the next step is to make it accessible via a URL. In Django, this is typically achieved using a URL router. The Django REST Framework (DRF) includes a URL router specifically designed for Viewsets, which automatically generates appropriate URLs for the typical actions (e.g., create, read, update, delete).

In this chapter, we will walk you through the process of registering your profile Viewset with this router, thus making your API accessible from the web.

---

1. Import Necessary Libraries and Modules:

To start, we'll need to import the required modules for URL configuration. If you remember, the `urls.py` file serves as the main routing file for Django projects.

“`python

from django.urls import path, include

from rest_framework.routers import DefaultRouter

from . import views

“`

Here:

- `path` and `include` are standard Django URL tools.

- `DefaultRouter` is a class from DRF used to automatically create routes for our Viewsets.

- `views` is our module containing the profile Viewset.

---

2. Create a Router Object:

Next, we need to instantiate our router. This router will be responsible for generating URLs for our Viewset.

“`python

router = DefaultRouter()

“`

---

3. Register the Viewset with the Router:

With the router created, we can now register our profile Viewset.

“`python

router.register(r’profile’, views.ProfileViewSet)

“`

Here, `r’profile’` specifies that the base URL for this resource will be “profile”. For instance, the URL to list all profiles will be ` http://yourapi.com/profile/ ` and the URL

for a specific profile might be ` http://yourapi.com/profile/1/ `.

---

4. Include the Router's URLs in the Main URL Configuration:

Finally, ensure that the generated routes are included in the main `urls.py` of the project.

“`python

urlpatterns = [

    path(”, include(router.urls)),

]

“`

This code includes all the routes generated by our router in our project's URL configuration.

---

5. Test the Configuration:

Now that our profile Viewset is registered with the URL router, it's a good idea to test and ensure everything works as expected.

1. Start the development server:

“`

python manage.py runserver

“`

2. Open your web browser or API client tool like Postman and navigate to ` http://localhost:8000/profile/ `. If everything is configured correctly, you should see the list of profiles, or at least an empty page if no profiles have been created yet.

Conclusion:

URL routing is essential for making your API endpoints accessible. The Django REST Framework's

`DefaultRouter` simplifies this process by automatically generating routes for your Viewsets. With the profile Viewset registered, we're one step closer to having a fully functioning Profiles API.

Note to the Readers:

Always ensure your URL patterns are clear and consistent. This will not only help you in the development process, but it'll also make your API more intuitive for other developers who might interact with it in the future.

Test creating a profile

Assets, Resources, and Materials required for this chapter:

- Python (Version 3.x): Ensure that you have Python installed. If not, download it from the official [Python website](https://www.python.org/downloads/). Python is the main language we're using to develop our REST API.

- Django and Django REST Framework: You should have these installed from the earlier chapters. If not, revisit Chapters 12 & 13 for installation details. These tools help us structure our web application and API.

- Postman: A popular tool for testing APIs. Download it from the [official website](https://www.postman.com/downloads/). We'll use this tool to make HTTP requests to our API and analyze the responses.

- Terminal or Command Prompt: For running server-side commands.

Objective: By the end of this chapter, you should be able to test the profile creation functionality of your REST API, ensuring that it behaves as expected.

---

1. Introduction

Before deploying any feature in an application, it's imperative to test it rigorously to ensure it functions as expected. In this chapter, we'll be using Postman to test the profile creation functionality of our REST API. Postman provides a user-friendly interface to send requests to our API, making it easier to test various scenarios.

---

2. Preparing the API for Testing

To start, ensure that your Django development server is running. If it's not, navigate to your project directory in the terminal and run:

"`bash

python manage.py runserver

"`

This will start your server, typically accessible at `http://127.0.0.1:8000/`.

---

3. Setting up Postman for Testing

1. Open Postman: After installing, launch Postman.

2. Create a New Request: Click on the "+" tab. This will open a new tab for creating a request.

3. Set Request Type to POST: Since we're testing profile creation, we'll be sending a POST request. From the dropdown left to the URL bar, select "POST".

4. Enter API URL: In the URL bar, type your API endpoint for creating a profile, e.g., ` http://127.0.0.1:8000/profiles/ `.

5. Set Headers: In many cases, your API might require specific headers (like Content-Type). Make sure it's set to "application/json".

6. Enter JSON Data: In the "Body" tab, select "raw", then enter the profile data in JSON format. Here's an

example:

“`json

{

“name”: “John Doe”,

“email”: “johndoe@example.com”,

“password”: “strongpassword”

}

“`

---

4. Sending the Request and Analyzing the Response

After setting up the request in Postman:

1. Send the Request: Click the “Send” button.

2. Analyze the Response: After sending the request, Postman will display the API’s response below. Here’s what you should check:

- Status Code: Ensure you’re getting a `201 Created` status, indicating successful profile creation.

- Response Body: Check if the API returns the expected data. For example, the new profile’s ID, name, and email (without the password for security reasons).

- Error Messages: If there’s an issue, the API should return an error message detailing what went wrong. This can be a validation error or server-side error.

---

5. Testing Edge Cases

It’s not enough to test a successful scenario; you must also ensure your API handles failures gracefully:

1. Missing Data: Try sending a request without one of the required fields, e.g., without an email. The API should return a `400 Bad Request` status with a clear error message.

2. Invalid Data: Enter invalid data, like an incorrectly formatted email. Again, the API should return a `400 Bad Request` status with an appropriate error message.

3. Duplicate Data: Try creating a profile with an email that already exists in the database. The API should prevent duplicate entries and send an error message.

Conclusion

Testing is a critical phase in software development. By ensuring that the profile creation functionality works as expected and handles failures gracefully, we're one step closer to having a robust and reliable API. In the next chapters, we'll expand our tests to other functionalities and ensure our API is ready for deployment.

Remember, "It's not a bug – it's an undocumented feature!" The more you test, the fewer "undocumented features" you'll have in your final product.

Create permission class

Assets, Resources, and Materials:

- Django: (To be installed via pip. Django is the core framework we are using to build our backend.)

- Django REST Framework (DRF): (To be installed via pip. DRF extends Django's abilities to work with APIs.)

- Text Editor (e.g., Atom): (Available for free download at the Atom website. We'll use this to write and view our code.)

- Terminal or Command Line Interface: (Built into your operating system. To run commands.)

- Django's permissions and authentication modules: (Comes with DRF installation. For creating and managing permissions.)

Introduction:

When working with REST APIs, it’s crucial to implement the necessary permissions to safeguard sensitive data and ensure proper data flow between the frontend and backend. In this chapter, we’ll focus on creating a custom permission class for our Profiles API using Django REST Framework.

What is a Permission Class?

Permission classes in Django REST Framework determine whether a request should be granted or denied access. DRF provides a set of built-in permission classes like `IsAuthenticated`, `IsAdminUser`, and `IsAuthenticatedOrReadOnly`. While these built-in classes are helpful, sometimes you’ll need more customized behavior, and that’s where creating your own permission classes comes into play.

Steps to Create a Custom Permission Class:

1. Create a New Permissions File:

In your `profiles_api` app directory, create a new file named `permissions.py`.

“`bash

touch profiles_api/permissions.py

“`

2. Setting up the Permission Class:

In `permissions.py`, begin by importing necessary modules and then define your custom permission class.

“`python

from rest_framework import permissions

class UpdateOwnProfile(permissions.BasePermission):

    “””Allow users to edit their own profile”””

    def has_object_permission(self, request, view, obj):

        “””Check user is trying to edit their own profile”””

```
        if request.method in
permissions.SAFE_METHODS:
         return True
        return obj.id == request.user.id
```

Here, we've created a permission called `UpdateOwnProfile` which checks if the request is a safe method (GET, HEAD, or OPTIONS). If it's a safe method, the request is granted permission, otherwise, it checks if the object the user is trying to modify belongs to them by comparing the object's id with the user's id.

3. Implementing the Permission:

Go to `views.py` in the `profiles_api` app directory. Here, you'll have to add the custom permission class to your `ProfileViewSet`.

First, import the custom permission at the top:

```python
from .permissions import UpdateOwnProfile
```

Then, within your `ProfileViewSet`, add the permission to the `permission_classes` attribute:

```python
permission_classes = [UpdateOwnProfile, ]
```

Now, any time a request is made to `ProfileViewSet`, the `UpdateOwnProfile` permission class will be used to check if the request should be granted or denied access.

Testing the Permission:

With the permission class now in place, it's important to test it. This can be done using tools like Postman or your browser. Try to update a profile other than your own and

observe the result. If the permission class is functioning correctly, you should get a forbidden response for such requests.

Conclusion:

Permission classes are a powerful way to handle permissions and access control in Django REST Framework. By understanding and using them effectively, you can create a secure environment for your data and API users. The custom permission class we just created ensures that users can only modify their own data, keeping our Profiles API secure and functional.

Add authentication and permissions to Viewset

Assets and Resources:

- Django REST Framework (DRF) Documentation: [Official DRF Documentation](https://www.django-rest-framework.org/)

  - For diving deep into authentication, permissions, and viewsets.

- Django Authentication: [Official Django Documentation] (https://docs.djangoproject.com/en/3.0/topics/auth/default/)

  - Understanding the built-in Django user authentication system.

- Postman or similar API testing tool: [Download Postman](https://www.postman.com/downloads/)

  - Used for sending requests and observing API behavior.

Introduction:

Django REST Framework (DRF) provides out-of-the-box tools for integrating authentication and permissions into your API. Ensuring that only authenticated users can make certain requests is crucial for the security and functionality of our Profiles API. In this chapter, we'll set up authentication and permissions for our profile viewset.

---

1. Setting up Authentication:

To ensure that only authenticated users can access certain views or carry out specific actions, we first need to set up authentication. DRF provides a number of authentication classes, including Basic Authentication, Token Authentication, and Session Authentication.

For our Profiles API, we'll use Token Authentication. Here's how to set it up:

a. Install the necessary package:

```
pip install djangorestframework[authtoken]
```

b. Add `'rest_framework.authtoken'` to your `INSTALLED_APPS` in `settings.py`:

```python
INSTALLED_APPS = [
    ...
    'rest_framework.authtoken',
    ...
]
```

c. Run migrations to add the token model to your database:

```
python manage.py migrate
```

“`

d. Add Token Authentication to your default authentication classes in `settings.py`:

“`python

REST_FRAMEWORK = {

    …

    ‘DEFAULT_AUTHENTICATION_CLASSES’: (

        ‘rest_framework.authentication.TokenAuthenticatio
n’,

    ),

    …

}

“`

Now, when a user logs in, they will be provided with a token which they must include in the header of their subsequent requests.

---

2. Setting up Permissions:

Permissions in DRF determine what type of request (GET, POST, PUT, etc.) a user is allowed to perform. Let’s define the necessary permissions for our profile viewset.

a. Define the default permission classes in `settings.py`:

“`python

REST_FRAMEWORK = {

    …

    ‘DEFAULT_PERMISSION_CLASSES’: (

        ‘rest_framework.permissions.IsAuthenticated’,

    ),

    …

}
“`

The above setting will ensure that only authenticated users can access the viewset. But, what if we want more granular permissions, like allowing anyone to view profiles but only authenticated users to edit them?

b. Create a custom permission:

In your app, create a file named `permissions.py` and add the following code:

“`python

from rest_framework import permissions

class UpdateOwnProfile(permissions.BasePermission):

    “””Allow users to edit their own profile”””

    def has_object_permission(self, request, view, obj):

        “””Check if user is trying to edit their own profile”””

        if request.method in
permissions.SAFE_METHODS:

            return True

        return obj.id == request.user.id

“`

Here, we’re allowing any request method that’s considered “safe” (i.e., GET requests). But for other methods like PUT or DELETE, we’re ensuring the profile being edited belongs to the authenticated user.

c. Add the custom permission to our Profile Viewset:

In your `views.py`, import the custom permission and add it to the profile viewset:

“`python

from .permissions import UpdateOwnProfile

class UserProfileViewSet(viewsets.ModelViewSet):

    …

```
    permission_classes = (UpdateOwnProfile, )

    …
```

---

3. Testing Our Setup:

Using Postman or another API tool, you can now send requests to your Profiles API to see the permissions and authentication in action. Ensure that:

- A user cannot edit another user's profile.

- Without a token, a user cannot edit any profile.

- Any user, even without logging in, can view profiles.

Conclusion:

Authentication and permissions are pillars of any robust API, ensuring data security and proper data access. By leveraging Django and Django REST Framework, you've been able to implement them with minimal hassle, focusing on what matters most - delivering a seamless user experience.

Test new permissions

Assets, Resources, and Materials Required:

- Django Project Environment: By this stage, you should have your Django project environment set up with the profiles API created.

- Django Rest Framework: Ensure you have Django Rest Framework (DRF) installed and configured as we'll be using DRF's test utilities.

- Postman (or any API testing tool): You can download Postman from [here](
https://www.postman.com/downloads/). It's a widely-used tool for testing APIs.

- Python Unit Testing Framework: Django comes bundled with Python's `unittest` framework which we will be utilizing.

- User Profile: A created user profile in your Django backend to test permissions against.

- ModHeader Extension: Ensure it's installed in your browser for setting token headers.

Purpose of Testing Permissions

Before diving into the testing process, let's understand why we're doing this. Permissions, in the context of an API, define who can do what. This ensures the integrity and security of data. By testing permissions, we aim to verify that:

1. Unauthenticated requests are denied access.

2. Authenticated requests from unauthorized users are denied access.

3. Authenticated and authorized requests are granted access.

---

1. Setting up Test Environment

First, we need to set up our environment for testing. Django comes with a built-in test database which gets created every time you run tests and gets destroyed after the tests are over.

Steps:

- Create a new file named `test_permissions.py` in your profiles app directory.

- Import necessary modules:

“`python

from rest_framework import status

from rest_framework.test import APITestCase

from django.contrib.auth import get_user_model

```
from .models import Profile
from rest_framework.authtoken.models import Token
```

---

2. Create Test Users

For our tests, we will create two users: One that we'll assign permissions to and another that we won't.

```python
class ProfilePermissionTests(APITestCase):
    def setUp(self):
        self.user1 = get_user_model().objects.create_user(
            username='user1',
            password='testpass123'
        )
        self.user2 = get_user_model().objects.create_user(
            username='user2',
            password='testpass123'
        )
        self.profile1 = Profile.objects.create(user=self.user1, … other_profile_details…)
        self.profile2 = Profile.objects.create(user=self.user2, … other_profile_details…)
        self.token1 = Token.objects.create(user=self.user1)
        self.token2 = Token.objects.create(user=self.user2)
```

3. Test Unauthenticated Requests

We need to ensure that an unauthenticated request doesn't have access to our profiles.

“`python

```
    def test_unauthenticated_permissions(self):
        response = self.client.get('/path_to_profiles_api/')
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

“`

---

4. Test Unauthorized Requests

Now, test the scenario where `user2` tries to modify the profile of `user1`. This should be denied.

“`python

```
    def test_unauthorized_user_permissions(self):
        self.client.credentials(HTTP_AUTHORIZATION='Token ' + self.token2.key)
        response = self.client.put('/path_to_profiles_api/profile1_id/', {…data…})
        self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)
```

“`

---

5. Test Authorized Requests

Here, `user1` will try to modify their own profile. This should be successful.

“`python

```
    def test_authorized_user_permissions(self):
        self.client.credentials(HTTP_AUTHORIZATION='Token ' + self.token1.key)
```

```
        response = self.client.put('/path_to_profiles_api/profile1_id/', {…data…})
        self.assertEqual(response.status_code, status.HTTP_200_OK)
```

---

6. Running the Tests

Now that you have set up your tests, it's time to run them.

In your terminal or command prompt, navigate to your Django project directory and run:

```bash
python manage.py test profiles
```

You should see the results of the tests on your terminal. Successful tests will indicate that the permissions are working as expected.

Conclusion

Testing permissions is crucial to ensuring the security and reliability of your API. Always remember to test after making changes to the permission configurations to guarantee the safety of your user data.

In the next chapter, we will look into adding a search functionality to the profiles API. This will allow clients to query the API for specific user profiles. Stay tuned!

Add search profiles feature

Assets, Resources, and Materials required for this chapter:

1. Django (Installation required: Can be acquired using pip, `pip install django==2.2`)

   - Use/Purpose: The main framework to build our web application.

2. Django REST Framework (Installation required: Can be acquired using pip, `pip install djangorestframework==3.9`)

   - Use/Purpose: An additional toolkit for Django to create RESTful APIs easily.

3. Database (You should already have it set up, either SQLite (comes with Django) or another database you've chosen.)

   - Use/Purpose: To store our user profile data.

4. Postman (Installation required: Can be downloaded from [Postman's official website](https://www.postman.com/downloads/))

   - Use/Purpose: A tool to test our search API.

Introduction

Searching is a crucial part of any application. As our userbase grows, we want our users to easily find and interact with other users' profiles. In this chapter, we will incorporate a search feature into our Profiles API. By the end of this chapter, you'll have an understanding of how to implement a search filter to locate profiles based on certain criteria.

Step-by-step Guide

1. Update the `views.py` File:

First, we need to modify our `profiles` viewset to support search.

"`python

from rest_framework import viewsets, filters

```
# … (other imports)
class UserProfileViewSet(viewsets.ModelViewSet):
    queryset = UserProfile.objects.all()
    serializer_class = UserProfileSerializer
    authentication_classes = (TokenAuthentication,)
    permission_classes = (UpdateOwnProfile,)
    filter_backends = (filters.SearchFilter,)
    search_fields = (‘name’, ‘email’,)
```

Here:

- `filter_backends` indicates which filtering backends we want to use. We’ve specified the `SearchFilter` provided by Django REST Framework.

- `search_fields` is a tuple indicating which model fields we want to search against. In this case, we’ve chosen the `name` and `email` fields.

2. Test the Search Functionality with Postman:

Launch Postman and set it up for a GET request. For instance, if you’re running the server locally, enter:

```
http://127.0.0.1:8000/api/profile/?search=john
```

This should return all user profiles where either the name or email contains the term “john”.

3. Optional: Implement a Custom Search Filter:

If the built-in search filter doesn’t meet your requirements, you can create a custom search filter.

```python
from django.db.models import Q
class CustomSearchFilter(filters.BaseFilterBackend):
```

```
    def filter_queryset(self, request, queryset, view):
        search_term = request.query_params.get('search', None)
        if search_term:
            queryset = queryset.filter(Q(name__icontains=search_term) | Q(email__icontains=search_term))
        return queryset
```

Then, update the `UserProfileViewSet` to use the custom search filter:

```python
class UserProfileViewSet(viewsets.ModelViewSet):
    # … (previous code)
    filter_backends = (CustomSearchFilter,)
```

4. Optimize with Database Indexing:

To ensure efficient searching, especially for larger datasets, it's recommended to add database indexes to fields that will be frequently searched.

In the `models.py`:

```python
class UserProfile(models.Model):
    name = models.CharField(max_length=255, db_index=True)
    email = models.EmailField(max_length=255, unique=True, db_index=True)
    # … (rest of the model fields)
```

Here, `db_index=True` indicates that a database index should be created for these fields.

After modifying the models, remember to create and apply migrations:

“`

python manage.py makemigrations

python manage.py migrate

“`

Conclusion

You’ve successfully added a search feature to your Profiles API! Now, users can easily find other profiles using search terms. As your application grows, features like this enhance user experience and make your application robust and user-friendly.

Test searching profiles

Assets, Resources, and Materials:

1. Django Development Environment: (By now, you’ve set up your environment using the previous chapters. If not, please refer to Chapter 5 - 16.)

2. Postman: (A popular tool to test APIs. Download and install from the [official Postman website](https://www.postman.com/downloads/).) - We will use Postman to test our API endpoints.

3. ModHeader Extension: (You’ve already installed this if you followed along. If not, get it from the web store for your browser.) - Used for setting token headers and simulating authenticated requests.

4. Sample Profile Data: (Please have some profiles in your database. This could be test data you’ve previously inputted or you can add some new profiles for this chapter.)

Introduction

In the previous chapters, we built a mechanism to allow users to search for profiles within our API. Searching is a core feature in any API where user-related data is stored. This chapter will focus on how to test the search functionality to ensure it works as expected.

Steps to Test Searching Profiles:

1. Ensure Profiles Exist: Before testing search functionality, ensure you have multiple profiles in your database. If not, create a few through the API or Django Admin.

2. Open Postman:

    - Start Postman and create a new request.

    - Set the request type to `GET`.

    - Use the endpoint URL for fetching profiles. It might look something like this: `http://127.0.0.1:8000/api/profiles/`.

3. Setting Up Headers:

    - Since our API is protected, set up the headers to include the token for authentication. Use the ModHeader extension to add your token.

4. Searching Profiles:

    - Add a query parameter to the end of your URL to search for profiles. This will be based on the search functionality you implemented. For instance, if you're searching for a user named "John", your URL might look like: `http://127.0.0.1:8000/api/profiles/?search=John`.

    - Send the request in Postman.

5. Analyze the Results:

    - After sending the request, look at the response. You should see profiles that match your search criteria. For our example, profiles containing the name "John" should appear.

- Ensure that the profiles returned are relevant to the search term.

6. Test Edge Cases:

- Case Sensitivity: Try searching with different letter cases, like “john”, “JOHN”, and “JoHn”, to ensure that the search is not case-sensitive.

- Partial Matches: Try searching for just a part of a name or email to see if profiles show up. For example, searching for “Jo” should still show profiles named “John”.

- No Matches: Search for a term that doesn’t exist in your profiles. You should get an empty result set.

7. Checking Error Responses: Make sure to handle cases where errors might occur. This could be due to an invalid token, a token that’s expired, or any other error. The API should return a clear error message to the user.

Testing is an essential part of the development process. By ensuring that the search functionality works as expected, we are taking another step towards providing a reliable and effective API for our users. Always remember to test every new feature and change to your API to ensure robustness and reliability.

Note: This is a basic overview of testing the search functionality for profiles. Depending on the complexity of your API and the search features you’ve implemented, there may be more test cases and considerations to take into account.

Section 11:

Create login API

Create login API viewset

Assets, Resources, and Materials for this chapter:

- Django REST Framework (DRF): To create our API viewsets. (To acquire: `pip install djangorestframework`)

- ModHeader Browser Extension: For setting up tokens in the header during testing. (To acquire: Install from Chrome Web Store or Firefox Add-ons Store)

- Postman: A popular tool for testing APIs. (To acquire: Download from the official Postman website)

Introduction:

Logging in is a crucial part of any web application. This chapter will walk you through the creation of a Login API Viewset. The objective is to let users send their credentials, validate them, and in response, send back a token to be used for authenticated requests.

---

1. Setting up the Login Serializer:

Before diving into the viewset, we need a serializer to validate the data coming in from the user. This will be a simple serializer with fields for the username and password.

“`python

from rest_framework import serializers

class LoginSerializer(serializers.Serializer):

    username = serializers.CharField()

    password = serializers.CharField(write_only=True)

“`

Here, `write_only=True` ensures that the password won’t be displayed when serialized.

---

2. Creating the Login API Viewset:

Now, we'll create a Viewset to handle the login.

First, import the necessary modules:

```python
from rest_framework import viewsets, status
from rest_framework.response import Response
from django.contrib.auth import authenticate
from rest_framework.authtoken.models import Token
```

Then, create the Viewset:

```python
class LoginViewSet(viewsets.ViewSet):
    serializer_class = LoginSerializer
    def create(self, request):
        """Handle user login and return auth token."""
        serializer =
self.serializer_class(data=request.data)
        if serializer.is_valid():
         user = authenticate(
            username=serializer.validated_data['username'
],
            password=serializer.validated_data['password']
         )
         if user:
            token, created =
Token.objects.get_or_create(user=user)
            return Response({'token': token.key},
status=status.HTTP_200_OK)
         else:
```

```
            return Response({'error': 'Invalid credentials'}, status=status.HTTP_401_UNAUTHORIZED)
        else:
            return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Explanation:

- The `LoginViewSet` uses a `LoginSerializer` to validate the incoming data.
- Within the `create` method, we try to authenticate the user. If successful, we retrieve or create a token for the user and return it.
- If the authentication fails, we return an error message.

---

3. Registering the Login API Viewset with URLs:

In your `urls.py`:

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from . import views
router = DefaultRouter()
router.register('login', views.LoginViewSet, basename='login')
urlpatterns = [
    path('', include(router.urls)),
]
```

Now, the endpoint `/login/` is ready to receive POST requests for user logins.

---

4. Testing the Login API:

We'll use Postman for this:

1. Open Postman.

2. Set the request type to `POST`.

3. Enter the URL: ` http://localhost:8000/login/ `.

4. Under 'Body', choose 'raw' and 'JSON (application/json)'.

5. Enter the following:

"`json

{

"username": "yourUsername",

"password": "yourPassword"

}

"`

Replace `yourUsername` and `yourPassword` with the credentials of a user in your database.

6. Click 'Send'. If successful, you'll receive a token in the response. Otherwise, you'll get an error message.

Conclusion:

By the end of this chapter, you've successfully set up a Login API Viewset that authenticates users and returns a token for authenticated requests. This token will be vital for making secure API calls in subsequent parts of your application.

Remember, security is paramount. Always keep your tokens secure, never expose them in client-side code, and make sure your API uses HTTPS in production to ensure encrypted communication.

In the next chapter, we'll discuss how to test our Login API, ensuring it functions as expected and remains secure against potential threats.

Test login API

Assets, Resources, and Materials for this Chapter:

1. Postman - A popular tool for testing APIs. (You can download it from the [official Postman website](https://www.postman.com/downloads/))

2. Login API Endpoint - The URL you've set up for your login functionality.

3. Django Dev Server - Make sure your development server is running.

4. Credentials - A set of username and password to test the login functionality. If you haven't created any users, you'd need to do so in the Django Admin.

Introduction

Once you've set up your login API, it's imperative to ensure it's functioning as expected. Testing is a critical phase in any development process, ensuring the correctness and reliability of the system. In this chapter, we will use Postman, a powerful API testing tool, to test our login API.

Setting Up Postman

1. Installation: If you haven't already, download and install Postman from the provided link.

2. Launching and Initial Setup:

   - Launch Postman after installation.

   - If you're a first-time user, you can skip the initial signup unless you want to save your collections to the cloud.

3. Setting Up A New Request:

   - Click on the '+' tab to create a new request.

- From the dropdown menu next to the URL input field, select ‘POST’ since our login API would typically use the POST method.

- Input your Login API Endpoint in the URL field (e.g., `http://127.0.0.1:8000/api/login/ `).

Testing the Login API

1. Adding Headers:

- In Postman, under the ‘Headers’ tab, you need to add a key-value pair:

  - Key: `Content-Type`

  - Value: `application/json`

This tells the server that you’re sending JSON data.

2. Inputting Body Data:

- Switch to the ‘Body’ tab in Postman.

- Select the ‘raw’ option and ensure ‘JSON’ is selected from the dropdown.

- Input a set of valid credentials. For instance:

  “`json

  {

  “username”: “sampleuser”,

  “password”: “samplepassword123”

  }

  “`

3. Sending the Request:

- Click the blue ‘Send’ button.

- Observe the response. If your API and server are functioning properly, you should receive a token as part of the response. This token represents a successful login and can be used for subsequent authenticated requests.

4. Negative Testing:

- It’s not just about positive scenarios (where everything works). You need to test what happens when things go wrong.

- Try inputting an incorrect password or username. The expected response should be an error, indicating invalid credentials.

- Input incomplete data, like omitting the password or username. The API should respond with an error indicating the missing fields.

5. Testing Token Expiry (Optional):

- Depending on your settings in Django Rest Framework, tokens might have an expiry time. If you’ve set up token expiration, wait for that duration and then try accessing an authenticated endpoint using the expired token. You should receive an ‘invalid token’ or ‘token expired’ response.

Conclusion

Testing is a continuous and ongoing process. Even after deploying your API, you should routinely perform tests, especially after making changes or updates. Using tools like Postman can significantly simplify and streamline the process, ensuring that your Login API is robust, secure, and reliable.

In the next chapter, we’ll explore how to set the token received upon successful login in the ModHeader extension, allowing for easy access to authenticated routes during further testing and development.

Note: This chapter focuses on manual testing using Postman. In a real-world scenario, especially for larger applications, you’d also want to implement automated tests for your APIs to ensure stability and consistency.

Set token header using ModHeader extension

Assets, Resources, and Materials for this Chapter:

- ModHeader Extension: This is a browser extension that allows you to manipulate browser request headers. It's available for both Google Chrome and Mozilla Firefox.

- Your API: Ensure you have your Django REST API up and running, especially the login API endpoint we developed in the previous chapters.

Introduction

After successfully logging in via our login API endpoint, a token will typically be returned to authenticate future requests. This token assures the server that the request comes from a legitimate, authenticated user. To send this token with each request, we need to include it in the request headers. One popular and straightforward way to do this during development and testing is by using the ModHeader browser extension.

In this chapter, we will go over how to use the ModHeader extension to set the token header, ensuring our API requests are authenticated.

---

1. Installing ModHeader

If you haven't already installed ModHeader:

1. Open your browser.

2. Go to the respective store:

   - Chrome users: Visit the [Chrome Web Store](https://chrome.google.com/webstore/)

   - Firefox users: Visit the [Firefox Add-ons site](https://addons.mozilla.org/)

3. Search for “ModHeader.”

4. Find the ModHeader extension in the search results and click “Add to Chrome” or “Add to Firefox.”

5. Follow the browser’s prompts to complete the installation.

---

2. Retrieving Your Token

Before we set our token in ModHeader, we first need to retrieve it:

1. Log in via your login API.

2. Once successfully logged in, the response will include an authentication token. Copy this token; we’ll need it for the next steps.

---

3. Setting the Token Header in ModHeader

1. Click on the ModHeader icon in your browser’s toolbar. This will open the ModHeader interface.

2. In the ModHeader interface, you’ll see two primary input fields: “Request Headers” and “Filter.”

3. Under “Request Headers,” click on the `+` button to add a new header.

4. For the header name, enter `Authorization`.

5. For the header value, enter `Token <Your-Token>`. Replace `<Your-Token>` with the token you copied from your login API response. It should look something like `Token 1234abcd5678efgh`.

Note: The space between “Token” and your actual token is essential.

---

4. Making Authenticated Requests

With the token header set in ModHeader:

1. Make any request to your API that requires authentication, like fetching a user’s profile or posting a

status update.

2. Observe that the request goes through successfully, indicating it's appropriately authenticated.

---

5. Removing or Disabling the Token Header

You might want to remove or disable the token header when testing unauthenticated requests or when done testing:

1. Open the ModHeader interface.

2. You can either:

- Click on the trash bin icon next to the `Authorization` header to remove it.

- Toggle the green switch at the top right to the off position to disable ModHeader temporarily. Toggle it back on when you need to use it again.

Conclusion

Setting a token header using ModHeader is a simple yet powerful way to test authenticated requests during development. Always ensure that you're using valid tokens and that you remove or disable them when necessary to prevent unauthorized access to your API. As a best practice, never share screenshots or videos of your ModHeader interface with the token visible to maintain security.

Section 12:

Create profile feed API

Plan profile feed API

Assets, Resources, and Materials:

- API Design Tool (e.g., Postman, Swagger): These tools help in planning and documenting APIs. They can be downloaded from their respective websites.

- Flowchart software (e.g., Lucidchart, Draw.io): Assists in designing the flow of how the API would function. Free versions are available online.

- Note-taking application (e.g., Notepad, Evernote): Useful for jotting down ideas and requirements.

Introduction

Planning is an essential part of any development process. For an API, it's crucial to ensure that the right endpoints are in place, with the appropriate methods and authentication checks. In this chapter, we will walk through planning the Profile Feed API, an essential part of our application.

What is a Profile Feed?

At a high level, a profile feed is similar to what you might find on social media platforms like Twitter or Instagram. It's a series of posts or status updates from a user. These posts can be created, read, updated, or deleted. In our case, the profile feed will showcase status updates made by users.

User Stories

Before we delve into the technicalities, let's define our user stories. User stories are an agile development tool used to capture product functionalities from the end user's perspective.

1. As a registered user, I want to post a status update so that others can see what I'm up to.

2. As a registered user, I want to view my past status updates to reminisce about past events.

3. As a registered user, I want to update a status in case I made an error.

4. As a registered user, I want to delete a status that is no longer relevant.

5. As a viewer, I want to view other users' status updates to stay updated on their activities.

Endpoints and Methods

Given our user stories, here's a tentative list of the endpoints and methods we'll need:

1. POST /feed/: Create a new status update.

2. GET /feed/{user_id}/: Retrieve all status updates for a particular user.

3. PUT /feed/{status_id}/: Update a specific status update.

4. DELETE /feed/{status_id}/: Delete a specific status update.

Data Structure

Next, we need to determine what data is associated with a status update. Here's a preliminary data structure:

- user_id: An identifier for the user who posted the status.

- status_id: A unique identifier for each status.

- content: The text content of the status update.

- timestamp: The date and time when the status was posted.

Authentication & Permissions

Ensuring that our API is secure is crucial. Given our user stories, here are some necessary permissions:

- All users can view status updates.

- Only the owner of a status update can update or delete it.

- Registered users can create a new status update.

Flow

Using a flowchart tool can help visualize the sequence of actions a user might take and how the API would handle these actions. For instance:

1. User logs in and receives an authentication token.

2. User makes a POST request to /feed/ with the status content.

3. The API checks the token, verifies it's valid, and associates the status update with that user.

4. The status update is saved to the database and returned to the user with a unique status_id.

5. For subsequent operations (like update or delete), the API checks both the token and if the user is the owner of the status update.

Conclusion

Planning an API requires a combination of understanding the end user's needs, defining clear endpoints and methods, and ensuring data integrity and security. With this plan in place for our Profile Feed API, we're ready to start developing! In the following chapters, we will dive into the actual implementation of our planned API.

Add new model Item

Assets, Resources, and Materials for this Chapter:

- Python (Purpose: Programming Language. You can download it from the official Python website.)

- Django (Purpose: Web Framework. Can be installed using pip - “pip install Django”.)

- Django’s ORM (Object-Relational Mapping) (Purpose: Database management. Included in Django.)

Introduction

In our journey of creating a robust REST API with Django, we are now at a crucial step where we will be introducing a new model to handle the feed items in our user profile feed. This “Item” model will store every feed or status update that a user posts.

Let’s dive into creating the model.

Step 1: Open Your Models.py File

Navigate to the `models.py` file in your Django app directory. This is where we will define our new model.

“`python

from django.db import models

from django.conf import settings

“`

Step 2: Define the Item Model

For our feed item, we’ll need the following fields:

1. `user_profile`: A foreign key linking to our user model. This will let us know which user posted the feed item.

2. `status_text`: A character field where the user’s status will be stored.

3. `created_on`: A date field that will automatically set the current date and time when a feed item is created.

Here’s how the code will look:

“`python

class Item(models.Model):

```
    user_profile = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    status_text = models.CharField(max_length=255)
    created_on = models.DateTimeField(auto_now_add=True)
    def __str__(self):
        return self.status_text
```

Step 3: Explanation of Model Fields

1. ForeignKey: This field is used to create a one-to-many relationship. Here, it's used to link each feed item to a user. The `on_delete=models.CASCADE` argument means that if a user profile is deleted, all of their feed items will also be deleted.

2. CharField: This is a field for storing character data. We've limited the status text to 255 characters.

3. DateTimeField: This field is used to store date and time data. The `auto_now_add=True` argument means the field will be set to the current date and time when a new feed item is created.

Step 4: Migrations

Once you've defined your model, the next step is to create a migration for it. Migrations are Django's way of propagating changes made to models (adding, deleting, or changing fields) into the database schema.

In your terminal, navigate to the root directory of your project and run:

```bash
python manage.py makemigrations
```

After this, run:

“`bash

python manage.py migrate

“`

This will apply the migration and create the corresponding table in your database.

Step 5: Register Model with Admin

To manage our `Item` model via the Django admin interface, you’ll need to register the model. Open `admin.py` in your app directory and modify it as follows:

“`python

from django.contrib import admin

from .models import Item

admin.site.register(Item)

“`

Now, when you log into the Django admin interface, you’ll be able to see, add, edit, and delete feed items.

Conclusion

With the addition of the `Item` model, our backend structure for handling user feeds is taking shape. As you proceed, you’ll be integrating this model with serializers and viewsets to create a full-fledged API for feed items. Remember, the heart of any application often lies in its data structures. By taking the time to thoughtfully design your models, you’ll set yourself up for success.

Create and run model migration

Assets, Resources, and Materials:

1. Python: (You can acquire Python from the [official website](https://www.python.org/downloads/). Used for running our Django project.)

2. Django: (Installed via pip with the command `pip install django`. Framework for our web application.)

3. Django REST Framework: (Installed via pip with the command `pip install djangorestframework`. Used for building APIs.)

4. Terminal or Command Prompt: (Comes pre-installed with your OS. Used for running commands.)

5. Text Editor: (I recommend Atom, which you can acquire from the [official website](https://atom.io/). Used for writing and editing code.)

Introduction:

Migration in Django is one of its shining features. In essence, migrations allow you to manage changes you make to your application's models (like adding a new field, deleting a model, etc.) and propagate these changes into your database schema. This chapter will guide you through creating a new model for our profile feed API and subsequently running the migration to reflect this new model in our database.

Step 1: Define the Model

Before we can run a migration, we need a model to migrate. Given that this chapter is about creating a profile feed API, let's create a model named `ProfileFeedItem`:

"`python

In models.py

from django.conf import settings

from django.db import models

```
class ProfileFeedItem(models.Model):
    user_profile = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    status_text = models.CharField(max_length=255)
    created_on = models.DateTimeField(auto_now_add=True)
```

Here's what each line does:

- `user_profile`: This is a foreign key linking each feed item to a user profile.
- `status_text`: This will store the actual text of our status updates.
- `created_on`: This captures the date and time the status was created.

Step 2: Prepare the Migration

Once you've defined your model, the next step is to tell Django to prepare a migration based on the changes you've made. This is done with the `makemigrations` command:

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

Upon running this command, Django will generate a migration file in the respective app's `migrations/` directory. This file will contain the steps to make the changes (in this case, creating a new table for our `ProfileFeedItem` model) in the database.

Step 3: Apply the Migration

With the migration file prepared, we can now apply it to update our database schema. Run the `migrate`

command:

“`bash

$ python manage.py migrate

“`

This will apply all pending migrations, including the one we just created for `ProfileFeedItem`. Once this command completes, the changes will be reflected in the database.

Step 4: Verify the Migration

To ensure that our migration has been successful, we can use Django’s built-in admin interface or directly inspect the database. For now, let’s register `ProfileFeedItem` with the admin site to easily visualize it:

1. In `admin.py`, add:

“`python

from .models import ProfileFeedItem

admin.site.register(ProfileFeedItem)

“`

2. Now, run your development server:

“`bash

$ python manage.py runserver

“`

3. Navigate to the Django admin site (`http://127.0.0.1:8000/admin/`) and log in. You should see `ProfileFeedItem` listed and be able to add and view items.

Conclusion:

Migration is an indispensable tool in a Django developer’s toolkit, allowing seamless transition from

model definition to a live, functioning database table. By now, you should have a good understanding of how to create and run migrations, bringing your `ProfileFeedItem` model to life.

Add profile feed model to admin

Assets, Resources, and Materials:

- Django Admin: (Included with Django, no additional setup required). Use: A built-in admin interface for managing database entries.

- Django Model: (Introduced earlier in the course). Use: Represents database schema and data structures.

- Code Editor: (Such as Atom, introduced in Chapter 3 & 4). Use: To write and modify your Python code.

- Running Development Server: Ensure your Django development server is running for testing purposes.

Introduction:

Django comes with a powerful admin interface that allows developers to interact with their models effortlessly. The interface gives a visual representation of your database, making CRUD operations (Create, Read, Update, Delete) simple. In this chapter, we'll walk through the steps to add our profile feed model to this admin interface.

Steps to Add Profile Feed Model to Django Admin:

1. Import Required Modules:

First, we need to make sure that our model and the Django admin module are imported in the `admin.py` file of our app.

“`python
from django.contrib import admin
from .models import ProfileFeedItem
“`

2. Register the Model:

After importing, the next step is to register the model with the Django admin site. By registering the model, we’re telling Django that this model should be available and manageable via the admin interface.

“`python
admin.site.register(ProfileFeedItem)
“`

Your `admin.py` should now look something like this:

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

3. Customizing the Admin Interface (Optional):

For a more tailored experience in the admin interface, you can create an admin class that provides configurations for how your model appears and behaves in the admin site.

“`python
class ProfileFeedItemAdmin(admin.ModelAdmin):
    list_display = [‘user_profile’, ‘status_text’, ‘created_on’]
    search_fields = [‘status_text’, ‘user_profile__name’]
admin.site.register(ProfileFeedItem, ProfileFeedItemAdmin)

“`

This will display the `user_profile`, `status_text`, and `created_on` fields as columns in the model’s list view. The `search_fields` attribute will add a search bar to the top of the page, allowing you to search feed items by `status_text` and the name of the associated `user_profile`.

4. Accessing the Admin Site:

To view and interact with the profile feed model via the admin interface:

1. Make sure your development server is running.

2. Open a web browser and navigate to: `http://127.0.0.1:8000/admin/`

3. Log in using your superuser credentials. (If you haven’t created a superuser, refer to Chapter 22).

4. After logging in, you should see your `ProfileFeedItem` listed under the app’s name.

5. Clicking on `ProfileFeedItem` will display a list of all feed items, and you’ll have options to add, modify, or delete entries.

Conclusion:

With your profile feed model now added to the Django admin interface, managing feed entries becomes much more straightforward. The admin interface is not just limited to CRUD operations. As your familiarity with Django grows, you’ll find that you can perform many other complex tasks, like exporting data, customizing the appearance, and adding in-line related models. These capabilities make Django’s admin interface an indispensable tool for developers.

What’s Next:

In the next chapter, we will delve into the process of serializing our profile feed model, which is an essential

step towards creating API endpoints for our application. Stay tuned!

Create profile feed item serializer

Assets, Resources, and Materials for this chapter:

- Django Rest Framework (DRF): This can be acquired by installing it using pip (`pip install djangorestframework`). We use DRF to create serializers, which allow complex data types, such as Django models, to be easily converted to Python data types that can be rendered into JSON.

- Models.py file: We should already have this from our Django app creation. This is where our `ProfileFeedItem` model resides.

- Serializers.py file: This is where we will create our serializer. If it doesn't exist, you'll need to create it.

Introduction:

Serializers allow complex data types, such as Django models, to be converted to Python data types that can be rendered into JSON. In essence, serializers allow us to easily convert database objects into a format that can be rendered into a response, and vice-versa. In this chapter, we will be creating a serializer for our `ProfileFeedItem` model, enabling us to easily handle profile feed data in our API.

Step-by-step guide:

1. Set Up Your Serializers File:

    If you haven't created a `serializers.py` file in your app directory, do it now.

“`bash

touch serializers.py

“`

2. Import Necessary Libraries and Models:

Open `serializers.py` and import the required modules and models.

“`python

from rest_framework import serializers

from .models import ProfileFeedItem

“`

3. Create the Serializer:

Now, let’s create the serializer class for the `ProfileFeedItem`. We’ll subclass the `serializers.ModelSerializer`.

“`python

class ProfileFeedItemSerializer(serializers.ModelSerializer):

class Meta:

model = ProfileFeedItem

fields = [‘id’, ‘user_profile’, ‘status_text’, ‘created_on’]

extra_kwargs = {‘user_profile’: {‘read_only’: True}}

“`

Here’s a breakdown of the code:

- model = ProfileFeedItem: This links our serializer to the `ProfileFeedItem` model.

- fields: This tuple defines which fields in the model we want to include in our serialized output (and also which fields we want to accept as input).

- extra_kwargs: Here, we make the ‘user_profile’ field read-only to ensure that users cannot change the profile

associated with a status update once it has been created.

4. Explanation of our Serializer:

- Our `ProfileFeedItemSerializer` will handle the conversion of our model instances into JSON.

- When we want to output a feed item in a response, we'll pass it to this serializer which will handle the conversion.

- Similarly, when we want to create a new feed item from received JSON data, this serializer will validate and parse the incoming data.

5. Testing the Serializer (Optional but recommended):

While it's not mandatory in production, testing serializers during the development phase can save a lot of headaches. Use Django's shell to test this.

"`bash

python manage.py shell

"`

Then, in the shell:

"`python

from your_app_name.serializers import ProfileFeedItemSerializer

from your_app_name.models import ProfileFeedItem

feed_item = ProfileFeedItem.objects.first() # or any other query to get a feed item

serializer = ProfileFeedItemSerializer(feed_item)

print(serializer.data)

"`

This should print out a serialized version of the feed item.

Conclusion:

Serializers are an integral part of Django Rest Framework, allowing us to easily manage complex data types and transform them into a more consumable format. By now, you should have a good understanding of creating a serializer for a specific model. This will serve as the foundation for our API views, which we will be diving into in the upcoming chapters.

Note: Remember to always check your code for errors and test thoroughly. It's crucial to ensure that our serializers work perfectly, as they're a foundational piece of our API's logic.

Next Up: In the following chapter, we will create the ViewSet for our profile feed item. This will be the heart of our API logic, utilizing the serializer we've just created.

Create ViewSet for our profile feed item

Assets, Resources, and Materials:

- Django REST Framework (DRF): This framework is required to use ViewSets and other helpful utilities that allow us to create APIs quickly. It can be installed using pip (`pip install djangorestframework`).

- Django Project & App: Ensure you have Django project & app set up. If you're following from previous chapters, you should have this ready.

- Profile Feed Model: This represents the database model we created for profile feed items in a previous chapter.

- Python Environment: Ensure your Python virtual environment is activated.

Introduction:

In the world of Django REST Framework, a `ViewSet` is a class-based view that does not provide any method handlers but inherits from basic class views like `.list()`, `.create()`, `.retrieve()`, `.update()`, and `.destroy()`. These provide the CRUD operations for our API.

For our Profile Feed API, we'll be using a `ViewSet` to manage the feed items, enabling functionalities like creating a new feed item, viewing all feed items, updating a feed item, and deleting one.

Step 1: Importing Required Modules

Before starting, we need to ensure that we have all the necessary imports:

“`python

from rest_framework import viewsets

from rest_framework.response import Response

from rest_framework import status

from .models import ProfileFeedItem

from .serializers import ProfileFeedItemSerializer

“`

Step 2: Create the Profile Feed Item ViewSet

Now, we'll create a ViewSet for our Profile Feed Item:

“`python

class ProfileFeedItemViewSet(viewsets.ModelViewSet):

    “””Handles creating, reading and updating profile feed items.”””

    serializer_class = ProfileFeedItemSerializer

    queryset = ProfileFeedItem.objects.all()

“`

In this `ProfileFeedItemViewSet`, we've done the following:

- Used the `ProfileFeedItemSerializer` to determine how the data will be transformed to and from JSON format.

- Specified the `queryset` which tells the ViewSet where to retrieve its data (in this case, all `ProfileFeedItem` instances).

Step 3: Adding the Profile Feed Item ViewSet to URLs

For our API to recognize this ViewSet, we need to wire it up with a URL. We'll use the `DefaultRouter` provided by DRF to automatically generate the URL patterns.

In your `urls.py`:

"`python

from django.urls import path, include

from rest_framework.routers import DefaultRouter

from . import views

router = DefaultRouter()

router.register(r'feed', views.ProfileFeedItemViewSet)

urlpatterns = [

    path('', include(router.urls)),

]

"`

The `DefaultRouter` automatically generates the URL patterns for our ViewSet, reducing the amount of code we need to write. The `r'feed'` argument specifies the name of the API endpoint.

Step 4: Testing the ViewSet

At this point, you should start your Django development server and navigate to the endpoint (typically `http://127.0.0.1:8000/feed/`) to view the list of all profile

feed items. You can also use tools like Postman or simply use your web browser to interact with your API.

Try creating a new feed item by posting data to the API or updating/deleting existing ones.

Conclusion:

Congratulations! You've successfully created a ViewSet for managing profile feed items in your API. With just a few lines of code, you've enabled CRUD functionalities. The power of Django REST Framework's ViewSets lies in the abstraction of common patterns, allowing developers to rapidly build robust APIs.

Remember, as your application grows, you might need to add custom functionalities to your ViewSets. DRF provides various mixins and methods to help you achieve that. Always refer to the official documentation to explore more advanced features.

Test Feed API

Assets, Resources, and Materials for this Chapter:

- Postman: (You can download Postman from the official website ` https://www.postman.com/downloads/ `. It's a popular tool for testing API endpoints.)

- ModHeader Browser Extension: (Available on the Chrome Web Store and Firefox Add-ons store. This extension allows you to modify HTTP request headers, which is handy for setting authorization tokens.)

- Python and Django setup: (Assuming you have already set up from previous chapters.)

- Our API's URL: (We'll use this to send test requests. It should have been defined in previous chapters.)

Introduction

Now that we've set up our profile feed API, it's crucial to test its functionality to ensure everything works as expected. Testing is a vital step in API development, as it helps identify potential issues and confirms the API behaves correctly. In this chapter, we'll use Postman to test our feed API.

Setting Up Postman

1. Install Postman: If you haven't already, download and install Postman from the link provided above.

2. Create a New Request: Launch Postman and click on the `+` sign to open a new tab. This will be our workspace for sending test requests.

Testing Our Feed API

1. Authorization:

- Before we can test our profile feed API, we need to be authorized. If your API uses token-based authentication, you should have received a token upon login or user registration.

- In Postman, under the `Headers` tab, set the key as `Authorization` and the value as `Token <your_token>`. (Replace `<your_token>` with your actual token.) Alternatively, if you are using the ModHeader browser extension, you can set the token there.

2. Testing `GET` Request:

- Set the request type in Postman to `GET`.

- Enter your API's feed URL (e.g., `http://localhost:8000/feed/ `).

- Click the `Send` button. You should see a list of feed items returned in the response, or an empty list if no feed items have been added.

3. Testing `POST` Request:

- Switch the request type to `POST`.

- Navigate to the `Body` tab in Postman and select `raw` and `JSON (application/json)`.

- Input a new feed item in the following format:

  “`json

  {

  “status_text”: “This is a test status!”

  }

  “`

- Click the `Send` button. You should receive a response indicating the feed item has been created.

4. Testing `PUT` Request:

- First, retrieve a feed item using the `GET` request to note its ID.

- Switch the request type to `PUT`.

- Modify the URL to target a specific feed item (e.g., ` http://localhost:8000/feed/1/ ` where `1` is the feed item’s ID).

- In the `Body` tab, input the updated status text:

  “`json

  {

  “status_text”: “This is an updated test status!”

  }

  “`

- Click the `Send` button. The response should show the updated feed item.

5. Testing `DELETE` Request:

- Use the same targeted URL as the `PUT` test.

- Switch the request type to `DELETE`.

- Click the `Send` button. You should receive a response indicating the feed item has been deleted.

Confirm by attempting another `GET` request for the same item; it should return a `404 Not Found`.

Handling Test Failures

In case any of the tests fail:

- Review the error message provided in the response. This can often give insights into what went wrong.

- Check your API code to ensure the endpoints are defined correctly and that authentication and permissions are set appropriately.

- Consult the Django logs for any server-side errors.

Conclusion

Thorough testing is essential for ensuring the reliability and robustness of your API. Remember to test after any changes or updates, and consider implementing automated testing in the future to streamline this process. With our feed API tests complete, we can confidently move forward with the next steps in our project.

Add permissions for feed API

Assets, Resources, and Materials:

- Django & Django REST Framework: If you haven't already installed these packages, they are essential for the development of our API. You can acquire them via pip by running `pip install django django-rest-framework`.

- Postman or a similar tool: A tool for API testing to see the permissions in action. You can download Postman from [their official website](https://www.postman.com/downloads/).

- Your existing project: Make sure you are working inside your existing Django project that has been covered in previous chapters.

- Custom User Model & Profile Feed Model: These should already be created in the earlier chapters.

Purpose of Permissions:

Permissions in Django REST Framework (DRF) are used to grant or deny access for different classes of users to different parts of the API. For our profile feed API, we want to ensure that only authenticated users can create, view, or modify feed items.

Step 1: Understanding Default Permissions

By default, DRF comes with a set of permissions that can be applied to our views. Some of the most commonly used permissions include:

- AllowAny: Gives unrestricted access to any user.

- IsAuthenticated: Grants access only to authenticated users.

- IsAdminUser: Grants access only to admin users.

- IsAuthenticatedOrReadOnly: Grants read-only access to unauthenticated users but full access to authenticated users.

For our feed API, we'll be using the `IsAuthenticated` permission to ensure only logged-in users can interact with the feeds.

Step 2: Setting Global Permissions

First, you can set the global permissions in the project settings to apply the `IsAuthenticated` permission to all views by default:

“`python

settings.py

```
REST_FRAMEWORK = {
    …
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
}
```

However, for this chapter, we will set the permissions explicitly in our view.

Step 3: Updating the Feed API View with Permissions

Navigate to your feed API view (created in previous chapters). Here, we'll explicitly set the permission class.

```python
# api/views.py (or wherever your profile feed view is located)
from rest_framework.permissions import IsAuthenticated
class ProfileFeedViewSet(viewsets.ModelViewSet):
    …
    permission_classes = (IsAuthenticated,)
    …
```

This ensures that only authenticated users can access this view.

Step 4: Testing the Permissions

Now that we've applied the permissions, it's essential to test and ensure they work correctly.

1. Unauthenticated Access Test: Use Postman or your preferred API testing tool. Try accessing the feed API

without logging in. You should receive a `401 Unauthorized` error.

2. Authenticated Access Test: First, log in with a user profile. Use the received token to access the feed API. You should be able to perform all the CRUD operations (Create, Retrieve, Update, Delete) without any issues.

Step 5: Custom Permissions (Optional)

For more granular control, DRF allows you to create custom permissions. For instance, if you wanted only the creator of a feed item to edit or delete it, you'd need a custom permission.

Here's a basic example:

“`python

from rest_framework.permissions import BasePermission

class IsOwner(BasePermission):

    ”””

    Custom permission to only allow owners of an object to edit or delete it.

    ”””

    def has_object_permission(self, request, view, obj):

        return obj.user_profile == request.user

“`

You can then add this permission to your view:

“`python

permission_classes = (IsAuthenticated, IsOwner,)

“`

Remember, the permissions are checked in the order they are listed. So, `IsAuthenticated` will be checked before `IsOwner`.

Conclusion:

Permissions play a crucial role in ensuring the security and functionality of our API. By now, you should have a good understanding of how to implement and test permissions in Django REST Framework. As always, ensure that you test all permissions thoroughly in both development and production environments to ensure the safety and integrity of your application.

Test feed API permissions

Assets, Resources, and Materials:

- Postman: (A popular API testing tool. It can be downloaded from [Postman's official website](https://www.postman.com/downloads/)). Use this tool to send HTTP requests and validate responses from our API.

- Your Django Backend: Ensure your Django server is running and that you've completed the setup for your profile feed API.

- API Endpoints: URLs to your feed API (usually something like ` http://localhost:8000/api/feed/ ` depending on your configuration).

Introduction

Testing is an integral part of software development. In the context of building APIs, testing ensures that endpoints are functioning correctly and providing the expected data in response to various requests. In this chapter, we're going to focus on testing the permissions of our feed API. Ensuring proper permissions is crucial for data privacy and integrity.

Goals:

- Test that unauthenticated users cannot access feed data.

- Test that authenticated users can access and post to the feed.

- Validate that users can only modify or delete their own feed data.

---

1. Set Up Postman for API Testing

1. Download and Install Postman: If you haven't installed Postman yet, download and install it from the link provided above.

2. Open Postman: Once installed, launch Postman.

3. Create a New Request: Click on the '+' tab to create a new request.

---

2. Test Unauthenticated Requests

Goal: Ensure that unauthenticated requests (those not providing a valid user token) cannot access the feed.

1. Set Up Request in Postman:

   - Set the request type to `GET`.

   - Input your feed API endpoint in the URL field (` http://localhost:8000/api/feed/ `).

2. Send the Request: Click on the `Send` button.

3. Review the Response: The API should return a `403 Forbidden` status, indicating the user does not have the proper permissions to view the feed without authentication.

---

3. Test Authenticated Requests

Goal: Authenticate a user and test that they can view and post to the feed.

1. Login to Retrieve a Token:

   - Set the request type to `POST`.

- Input your login API endpoint (`http://localhost:8000/api/login/ `).

- In the `Body` tab, input a valid username and password to obtain a token.

2. Add Token to Request Headers:

- Click on the `Headers` tab in Postman.

- Add a new header with `Key` as `Authorization` and `Value` as `Token <your_token_here>`, replacing `<your_token_here>` with the token you obtained.

3. Test GET Request with Authentication:

- Set the request type back to `GET`.

- Input your feed API endpoint in the URL field.

- Click `Send`. You should now receive a `200 OK` status, and the feed data should appear in the response body.

4. Test POST Request (Adding to Feed) with Authentication:

- Set the request type to `POST`.

- Input the same feed API endpoint.

- In the `Body` tab, set the data you wish to add to the feed.

- Click `Send`. You should receive a `201 Created` status, and the new feed data should be echoed back in the response.

---

4. Test User-Specific Permissions

Goal: Ensure users can only modify or delete their own feed data.

1. Create a Second User (if not already done): This user will be used to test that they can't modify another user's feed data.

2. Test Modifying Someone Else's Feed Data:

- Log in as the second user and obtain their token.

- Attempt to modify (using a `PUT` or `PATCH` request) or delete (using a `DELETE` request) feed data created by the first user.

- The API should return a `403 Forbidden` status, indicating the second user does not have the proper permissions to modify another user's feed data.

Conclusion

By the end of this chapter, you should have successfully tested various permissions related to your feed API. Properly testing permissions ensures data integrity and privacy, essential components of a trustworthy backend system. Always remember, building the API is just one part of the process; rigorous testing is what ensures it works as expected in all scenarios.

Restrict viewing status updates to logged in users only

Assets and Resources Required:

1. Django & Django REST Framework (DRF): (To build and test our API. If not already installed, you can get them via pip with `pip install django djangorestframework`.)

2. Postman or Browser: (For testing our API views and endpoints. Postman can be downloaded from [here](https://www.postman.com/downloads/).)

3. Django's built-in authentication: (For handling user authentication, which is included with the Django package.)

4. Profile Feed API built in previous chapters.

Introduction:

Before we jump into the coding part, let's understand the importance of this feature. One of the most common requirements for modern web applications is to restrict certain content or features to authenticated users. For social platforms or user-based platforms, status updates or feeds are often personal. Ensuring that these updates are only available to logged-in users keeps user data protected.

Step 1: Update the ViewSet Permissions

1. Open up the views.py where you have defined the `ProfileFeedItemViewSet`.

2. Add the required imports at the top of the file:

   "`python

   from rest_framework.permissions import IsAuthenticated

   "`

3. Update the `ProfileFeedItemViewSet` to include our new permission:

   "`python

   class ProfileFeedItemViewSet(viewsets.ModelViewSet):

       …

       permission_classes = (IsAuthenticated,)

   "`

What we've done here is specify that the `IsAuthenticated` permission is required for all operations in the `ProfileFeedItemViewSet`.

Step 2: Test the Restriction

Before moving forward, let’s ensure that our restriction is working:

1. Open up Postman or your browser.

2. Attempt to access the Profile Feed API without being logged in. You should receive a `403 Forbidden` error.

3. Now, log in with a user (using the login API we previously set up).

4. Use the token you received upon login to authenticate your request.

5. Access the Profile Feed API again. This time, you should be able to view the status updates since you’re authenticated.

Step 3: Customize the Permission

While the `IsAuthenticated` permission is a great start, sometimes you might want to add custom logic to your permissions. For instance, you might want to restrict certain actions to the owner of a profile.

To achieve this:

1. Create a new permissions file if you don’t already have one (typically named `permissions.py` in the app directory).

2. Define a new permission class:

```python
from rest_framework.permissions import BasePermission

class IsOwnerOrReadOnly(BasePermission):
    """
    Custom permission to only allow owners to edit their own profiles.
    """

    def has_object_permission(self, request, view, obj):
```

```
        # Read-only permissions are allowed for any request
        if request.method in ['GET']:
        return True
        # Write permissions are only allowed for the owner
        return obj.user == request.user
```

3. Update the `ProfileFeedItemViewSet` in `views.py` to use this new permission:

```python
permission_classes = (IsAuthenticated, IsOwnerOrReadOnly,)
```

Step 4: Additional Considerations

While we've restricted the view to authenticated users, consider the following for enhanced security:

- Pagination: Limit the number of returned results to prevent data scraping.

- Rate Limiting: Limit how frequently an endpoint can be accessed to prevent abuse.

- Secure Tokens: Regularly rotate and expire authentication tokens.

Conclusion:

With a few lines of code, we've managed to add a critical security layer to our application. Regularly testing and revisiting your app's permissions ensures that you maintain a robust and user-trusted platform. The next time you think of features, always remember that sometimes it's not about what features you add, but what you restrict that counts.

Test new private feed

Assets and Resources:

- Python: Programming language used for the backend development of the REST API.

  - How to acquire: Download and install from the [official website](https://www.python.org/downloads/).

  - Use: Core language for building our REST API.

- Django: High-level web framework for Python.

  - How to acquire: Installed via pip (`pip install django`).

  - Use: Framework for building our web application and API.

- Django REST Framework: Toolkit for building Web APIs in Django.

  - How to acquire: Installed via pip (`pip install djangorestframework`).

  - Use: Provides tools to build our REST API.

- Postman or ModHeader: Tools to test API endpoints.

  - How to acquire: Download [Postman](https://www.postman.com/downloads/) or install the [ModHeader extension](https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkpgkljpfnnfcklj) from the Chrome Web Store.

  - Use: To test API endpoints, send requests, and view responses.

- SQLite: Default database used by Django for development purposes.

  - How to acquire: Comes pre-configured with Django.

  - Use: Store and retrieve data for our API.

Introduction

Having implemented a new feature to restrict viewing status updates to logged-in users only, it is essential that we now test this functionality. Ensuring privacy and security is a significant part of any application, especially in the realm of user content and personal information. In this chapter, we will conduct tests to verify that our private feed system functions as expected.

Step 1: Setup

Ensure that your Django server is running by navigating to your project directory in your terminal or command prompt and running:

“`bash

python manage.py runserver

“`

Ensure that you have either Postman or ModHeader ready for testing the API endpoints.

Step 2: Testing Without Authentication

1. Open Postman.

2. Set the request method to GET.

3. Enter the endpoint URL for fetching the profile feed.

4. Click on Send.

Expected Result:

You should receive a 403 Forbidden error. This indicates that non-authenticated users cannot access the feed.

Step 3: Testing With Authentication

To ensure that only logged-in users can view the profile feed, we need to test with an authenticated request:

1. Register a new user or use an existing user’s credentials.

2. Login with the user’s credentials and obtain the authentication token.

3. In Postman:

- Set the request method to GET.
- Enter the endpoint URL for fetching the profile feed.
- In the Headers section, add a new header:
  - Key: Authorization
  - Value: Token YOUR_TOKEN_HERE (Replace `YOUR_TOKEN_HERE` with the authentication token you obtained after logging in).

4. Click on Send.

Expected Result:

You should receive a 200 OK response along with the list of status updates. This indicates that authenticated users can successfully access the private feed.

Step 4: Cross-User Privacy Test

To ensure that users can only view their own private feeds and not feeds of other users:

1. Create or use a second user account.

2. Log in with this account to obtain its authentication token.

3. In Postman, using the same GET request as above, replace the original token in the Authorization header with the second user’s token.

4. Click on Send.

Expected Result:

The returned feed should only contain status updates from the second user and not the original user. This confirms that each user can only access their own private feed.

Step 5: Logout Test

1. Logout from the user account.

2. Using Postman, try accessing the private feed with the same GET request and token.

Expected Result:

You should receive a 403 Forbidden error again. This confirms that once a user is logged out, they can't access the private feed even if they have a previously valid token.

Conclusion

By the end of these tests, you should be confident in the functionality and security of the private profile feed. It's crucial always to test new features thoroughly, especially those related to user privacy and authentication, to ensure the robustness and reliability of your application. Always remember, a well-tested application is a reliable application.

Section 13:

Deploying our API to a server on AWS

Introduction to deploying our app to AWS

Assets, Resources, and Materials for this Chapter:

1. AWS Account: You will need an AWS account. If you don't have one, sign up at [Amazon Web Services](https://aws.amazon.com/) (Use: To access and utilize AWS services).

2. AWS EC2 Documentation: [Official Documentation](https://docs.aws.amazon.com/ec2/index.html) (Use: As a reference guide and for deep diving into specific functionalities).

3. AWS CLI: [Installation Guide](https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html) (Use: To manage AWS services from the command line).

4. SSH Client: Depending on your OS, you may already have one installed. If not, you can use tools like PuTTY for Windows or the built-in Terminal for macOS and Linux (Use: To connect to AWS EC2 instances).

---

1. Introduction

Deploying an application to the cloud can initially seem like a daunting task, especially if you're unfamiliar with cloud concepts or specific cloud providers like AWS. However, AWS has made deploying applications relatively straightforward, and this chapter aims to introduce you to the fundamental concepts and steps you'll take to deploy your Django REST API to an AWS server.

By the end of this chapter, you'll have a clear understanding of:

1. Why we choose AWS for deployment.

2. The primary AWS service we'll be using: Amazon EC2 (Elastic Compute Cloud).

3. Key concepts related to AWS EC2.

---

2. Why Choose AWS?

There are numerous cloud providers available today, including Google Cloud Platform (GCP), Microsoft Azure, and others. So, why would one opt for AWS?

1. Maturity and Coverage: AWS, launched in 2006, is one of the earliest and most mature cloud platforms. It

offers a wide range of services and tools that cater to nearly every need.

2. Scalability: AWS's services, especially EC2, are built to scale. Whether you're serving ten users or ten million, AWS can handle it.

3. Security: AWS provides various tools and best practices to ensure that applications and data are secure.

4. Cost-effective: With AWS's pay-as-you-go pricing model, you only pay for the resources you use.

---

3. Introduction to Amazon EC2 (Elastic Compute Cloud)

Amazon EC2 is a web service that provides resizable compute capacity in the cloud. In simpler terms, it allows you to run virtual servers and scale computing capacity based on your requirements.

Key Concepts Related to AWS EC2:

1. Instances: Virtual servers in the Amazon EC2 environment.

2. Amazon Machine Images (AMI): Pre-configured packages required to launch an instance. This includes the OS, server, application server, and applications.

3. Instance Types: Define the hardware of the host computer used for the instance. It affects aspects like memory, CPU, storage, etc.

4. Security Groups: Act as virtual firewalls controlling the traffic for one or more instances.

5. Key Pairs: Secure login information for instances. It comprises a public key that AWS stores and a private key file you store.

---

4. Preparing for Deployment

Before we delve into the deployment process in the subsequent chapters, it's essential to get acquainted

with a few prerequisites:

1. Understand your Application: Know the requirements of your application. Does it need a specific OS? What are the hardware requirements?

2. Choose the Right EC2 Instance: AWS offers various instance types tailored for different needs. Based on your application's requirements, select an appropriate one.

3. Security: Make sure to understand AWS's security best practices. Misconfigurations can expose your application to unnecessary risks.

---

5. Conclusion

AWS provides a robust platform for deploying web applications like our Django REST API. In this chapter, we've introduced the reasons to choose AWS and given an overview of the EC2 service, which we'll be using to deploy our app.

In the upcoming chapters, we'll delve deeper into the specifics of setting up our application on AWS, securing it, and ensuring that it runs smoothly to serve our users.

Note to Readers : AWS and its services evolve over time, with new features added regularly. It's a good practice to refer to AWS's official documentation for the most current and in-depth information.

Add key pair to AWS

Assets, Resources, and Materials for this Chapter:

- AWS Management Console (Acquire by signing up on the AWS official website)

- Computer with SSH client (Inbuilt in macOS and Linux. For Windows, use software like PuTTY)

Introduction:

Before we can deploy our Django REST API to AWS, one of the essential steps is setting up a Key Pair. AWS uses public-key cryptography to encrypt and decrypt login information. A Key Pair consists of a public key, which AWS uses to encrypt data, and a private key, which you use to decrypt it. In simpler terms, think of it as a unique digital key that lets you access your server securely.

This chapter will guide you through creating and downloading a Key Pair on AWS, which you'll later use to SSH into your server instance.

---

1. Sign in to the AWS Management Console

Start by navigating to the AWS Management Console. If you don't have an account, you'll need to sign up. Once logged in, select the region in the top right corner where you want to deploy your services.

---

2. Navigate to the EC2 Dashboard

From the AWS Management Console, find the EC2 service either by typing "EC2" into the search bar or locating it in the "Recently visited services" section.

---

3. Accessing the Key Pair Section

On the EC2 Dashboard, find the "Key Pairs" option under the "Network & Security" section in the sidebar.

---

4. Create a New Key Pair

- Click on the "Create key pair" button.

- Provide a name for your key pair, e.g., "MyDjangoRESTAPIKey".

- From the "File format" dropdown, select the appropriate file format based on your OS:

- `.pem` for Linux and macOS
- `.ppk` for Windows (if you're using PuTTY)

- Click on the "Create key pair" button.

---

5. Download and Secure the Private Key

Once you've clicked on "Create key pair", the private key file will automatically download to your computer. This private key is essential to access your server, so treat it with utmost care.

- Never share your private key.

- Move it to a secure location on your computer.

- If you're on macOS or Linux, you might also need to set the right permissions for the key using the command:

"`bash

chmod 400 path_to_your_key.pem

"`

---

6. What if you lose the Private Key?

It's important to understand that if you lose access to your private key, AWS cannot help you retrieve it. In such cases, you'd need to create a new key pair and associate it with your EC2 instance or launch a new instance using the new key pair.

Conclusion:

You have now successfully created and downloaded your Key Pair from AWS. This Key Pair will play a pivotal role when we move on to creating our EC2 server instance and connecting to it securely. In the subsequent chapters, we'll dive deeper into setting up our server on AWS, using the Key Pair we've just generated. Remember, always keep your private key confidential to ensure the security of your server.

Create EC2 server instance

Assets, Resources, and Materials:

1. AWS Account: An account on Amazon Web Services. If you don't have one, you can sign up [here](https://aws.amazon.com/). This will allow you to access the EC2 service and create server instances.

2. Key Pair: This was created and discussed in Chapter 65. It's used for secure SSH access to your EC2 instance.

3. SSH Client: Software to securely connect to your EC2 server instance. Popular choices include `PuTTY` for Windows or the built-in `ssh` command for macOS and Linux.

Introduction:

In this chapter, we'll guide you through creating an EC2 (Elastic Compute Cloud) server instance on AWS. Amazon EC2 offers scalable compute capacity in the cloud, making web-scale computing easier for developers. This will be the remote server where we'll deploy our Django REST API.

Step-by-Step Guide to Creating an EC2 server instance:

1. Sign in to the AWS Management Console:

Log in to your AWS account. If you're new to AWS, you'll need to sign up and provide billing details. Don't worry; AWS offers a free tier for EC2, which allows you to explore the service without incurring costs.

2. Navigate to EC2:

From the AWS Management Console, click on `Services` and then select `EC2` under the `Compute` category.

3. Launch a New Instance:

Click the `Launch Instance` button to start the process.

4. Choose an Amazon Machine Image (AMI):

AMIs are pre-configured templates for instances. For our purpose, select the `Ubuntu Server` as it's a popular and well-supported choice. Ensure you pick the latest stable version.

5. Choose an Instance Type:

For the purpose of this tutorial, you can choose the `t2.micro` instance type, which is eligible for the AWS free tier. It's suitable for development and testing purposes. Later, based on your production needs, you can choose a more powerful instance.

6. Configure Instance:

- Number of instances: 1 (since we are setting up just one server)

- Network: Default VPC (Virtual Private Cloud)

- Subnet: Choose the default for now. A subnet represents a range within your VPC's IP address block.

- Ensure `Auto-assign Public IP` is set to `Enable`. This gives your instance a public IP address, which you can use to access it from the internet.

7. Add Storage:

The default storage (root volume) is 8GB on General Purpose SSD. This should be sufficient for our tutorial. However, make sure to adjust this based on the needs of your application in a real-world scenario.

8. Add Tags:

Tags can be useful for billing and organization. As an example:

- Key: `Name`

- Value: `DjangoRESTAPI-Server`

9. Configure Security Group:

A security group acts as a firewall that controls the inbound and outbound traffic to the instance. For our

purpose:

- Create a new security group.

- Name it something descriptive, e.g., `DjangoRESTAPISG`.

- Add rules:

  - `SSH`: Allows secure access to your server.

  - `HTTP`: Allows HTTP traffic.

  - `HTTPS`: Allows HTTPS traffic.

- For the source, select `Anywhere`. This allows connections from any IP. (In a production scenario, you'd limit this to known IPs for security reasons.)

10. Review and Launch:

Review your settings. When satisfied, click on the `Launch` button. A prompt will appear asking for a key pair. Use the key pair you created in Chapter 65. This will allow you to securely SSH into your server.

11. Wait for Initialization:

After launching, it takes a few minutes for the EC2 instance to initialize. Once the instance state turns "running", you can connect to it using its Public IP.

Conclusion:

Congratulations! You've successfully created an EC2 server instance on AWS. In the next chapters, we'll guide you through deploying our Django REST API onto this server, making it accessible from anywhere in the world. Remember to keep your key pair secure, as it's your gateway to accessing your server.

Add deployment script and configs to our project

Assets, Resources, and Materials:

- Amazon EC2 (Elastic Compute Cloud): This is a web service that provides scalable compute capacity in the cloud. You can sign up for an AWS account [here](https://aws.amazon.com/) if you don't have one.

- Deployment script: A set of instructions in a file to automatically deploy our project to a server.

- `requirements.txt`: Lists all the Python dependencies required for our project.

- SSH (Secure Shell): A cryptographic network protocol for securely starting a session on a remote server. Tools like [PuTTY](https://www.putty.org/) for Windows or the built-in SSH client for macOS and Linux can be used.

- `gunicorn`: A Python WSGI HTTP server for UNIX. This will be our application server for Django in production.

- AWS CLI (Command Line Interface): A tool provided by AWS to manage AWS services. You can download it [here](https://aws.amazon.com/cli/).

- Virtualenv: A tool to create isolated Python environments.

- Configuration files: Essential for setting environment-specific configurations, secrets, and variables.

---

1. Introduction:

Deployment is the stage where our local development work gets presented to the world. As we're deploying on AWS EC2, we need to automate the process to ensure consistent and rapid deployments.

---

2. Setting Up the Environment:

Before we move on to writing our deployment script, let's ensure our environment is ready:

Make sure AWS CLI is installed.

You can install it with:

```
pip install awscli
```

Ensure you've set up AWS CLI with the credentials.
Run:

```
aws configure
```

You'll be prompted to input your AWS Access Key, Secret Key, region, and default output format.

---

3. Create `requirements.txt`:

Our EC2 instance needs to know which Python packages to install. Inside your project directory, run:

```
pip freeze > requirements.txt
```

This creates a file with all your Python dependencies.

---

4. Writing the Deployment Script:

Let's create a script named `deploy.sh`. This script will handle the deployment process for us.

```bash
#!/bin/bash
# Navigate to our project directory
cd /path/to/your/django/project
# Activate our virtual environment
source venv/bin/activate
# Install necessary Python packages
pip install -r requirements.txt
```

```
# Collect static files
python manage.py collectstatic —noinput
# Finally, run our application using gunicorn
gunicorn your_project_name.wsgi:application —bind 0.0.0.0:8000
```

Don’t forget to make this script executable:

```
chmod +x deploy.sh
```

---

5. Configuration Files:

While deploying, we need different configurations compared to the development environment. This is where configuration files come into play.

Create a `prod_settings.py` in your Django project’s settings directory . This file will contain production-specific configurations. For instance, `DEBUG` should be set to `False`.

Ensure you have secret keys and database configurations hidden, possibly using environment variables or secret management tools provided by AWS like AWS Secrets Manager.

---

6. Uploading Our Script and Configs to AWS:

Now, we will upload our deployment script and configurations to our EC2 instance. Make sure you’ve already set up your EC2 instance and have your `.pem` key file ready.

Using SSH:

```
```

```
scp -i path_to_your_key.pem deploy.sh ubuntu@your_ec2_ip:/path/on/server

scp -i path_to_your_key.pem requirements.txt ubuntu@your_ec2_ip:/path/on/server

scp -i path_to_your_key.pem -r your_django_project ubuntu@your_ec2_ip:/path/on/server
```

“`

---

7. Conclusion:

By now, you should have your deployment script and configurations set up on your EC2 instance. In the next chapter, we’ll see how to execute this script and get our Django project live!

Deploy to server

Assets, Resources, and Materials:

- AWS EC2 Instance: This is the virtual server where we’ll deploy our Django project. You can create and manage your EC2 instances through the AWS Management Console. (Acquisition: Sign up or log in to your AWS account and navigate to the EC2 dashboard).

- Git: We’ll use Git for version control. (Acquisition: Install from git-scm.com).

- Virtual Environment: We’ll need our Python virtual environment to manage our project’s dependencies. (Acquisition: Use `pip install virtualenv`).

- Requirements.txt: A file containing a list of all the dependencies our project needs. This will be created using `pip freeze > requirements.txt` in our project directory.

- SSH Key Pair: Secure keys for connecting to your EC2 instance. This is generated from AWS when you create an EC2 instance.

Introduction

Deploying our API to a server, especially on AWS, is an essential step to make our application accessible to users. In this chapter, we'll walk through the process of deploying our Django REST API on an AWS EC2 instance.

---

1. Prepare Your Django Project for Deployment

Before deploying, ensure the following:

- Your Django project runs without errors locally.

- `DEBUG` is set to `False` in settings.py.

- You have added your server's IP address to `ALLOWED_HOSTS` in settings.py.

- You have generated the `requirements.txt` using the command:

“`bash

pip freeze > requirements.txt

“`

---

2. Launch an EC2 Instance

1. Log into the AWS Management Console and navigate to the EC2 dashboard.

2. Click on “Launch Instance” and select your preferred OS (For this guide, we'll use Ubuntu 18.04).

3. Choose an instance type (t2.micro is free for new AWS users within the free tier).

4. Click “Next” until you reach “Configure Security Group”. Here, add a rule to allow HTTP (Port 80) and HTTPS (Port 443).

5. Launch the instance and create a new key pair if you haven't. Download the key pair (This will be a .pem file) and keep it safe.

---

3. Connect to Your EC2 Instance

1. Navigate to your EC2 dashboard and find the “Public IP” of your instance.

2. Open your terminal and navigate to where your .pem file is located.

3. Connect to your instance using SSH:

“`bash

ssh -i “your-key.pem” ubuntu@your-ec2-public-ip

“`

---

4. Set Up the Environment on EC2

1. Update the package list and install essential packages:

“`bash

sudo apt update && sudo apt upgrade -y

sudo apt install python3-pip python3-venv git -y

“`

2. Clone your project from GitHub or transfer your project files to the server.

3. Navigate to your project directory and create a virtual environment:

“`bash

python3 -m venv venv

“`

4. Activate the virtual environment:

“`bash

source venv/bin/activate

“`

5. Install project dependencies:

“`bash

```
pip install -r requirements.txt
```

---

5. Set Up a Web Server

We'll use Gunicorn as our application server and Nginx as a reverse proxy:

1. Install Gunicorn in your virtual environment:

```bash
pip install gunicorn
```

2. Test if Gunicorn can serve your project:

```bash
gunicorn your_project_name.wsgi:application —bind 0.0.0.0:8000
```

3. Install Nginx:

```bash
sudo apt install nginx -y
```

4. Configure Nginx to forward requests to Gunicorn. Create a new configuration in `/etc/nginx/sites-available/your_project_name` and add the following:

```
server {
    listen 80;
    server_name your-ec2-public-ip;
    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
```

```
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
    }
}
```

5. Create a symlink for this configuration in `sites-enabled` and restart Nginx:

```bash
sudo ln -s /etc/nginx/sites-available/your_project_name /etc/nginx/sites-enabled
sudo nginx -t && sudo systemctl restart nginx
```

---

6. Secure Your Application with SSL

1. Add HTTPS to the allowed protocols in your Django settings.

2. Install Certbot for a free SSL certificate from Let's Encrypt:

```bash
sudo apt install certbot python3-certbot-nginx -y
```

3. Request a certificate:

```bash
sudo certbot —nginx
```

Follow the prompts and Certbot will automatically configure Nginx to use SSL.

---

7. Finalize Deployment

1. Use `systemd` to ensure Gunicorn runs as a service. This means it'll start automatically even after server reboots.

2. Keep your application updated by regularly pulling changes from your repository and restarting Gunicorn and Nginx as needed.

Congratulations! Your Django REST API is now live on AWS! Ensure to monitor your application and regularly backup your server and database.

Update allowed hosts and deploy changes

Assets, Resources, and Materials required for this chapter:

- AWS Management Console Access (How to Acquire: Sign up or log in at [https://aws.amazon.com/] (https://aws.amazon.com/))

- Python (How to Acquire: [Download and install Python](https://www.python.org/downloads/))

- Django project with your REST API (As built in previous chapters)

- A code editor (For the purpose of this book, we've been using Atom)

Introduction:

When deploying a Django project, especially to a public-facing server, it is vital to ensure that your application's settings are properly configured for security. One of these settings is `ALLOWED_HOSTS`. In Django, the `ALLOWED_HOSTS` setting is a security measure to prevent HTTP Host header attacks. Before we move to the actual deployment, let's address this setting.

Understanding `ALLOWED_HOSTS`:

Django uses this to match the `Host` header against a list of strings representing the `host/domain names` that this Django site can serve. It is a safety mechanism to prevent a particular type of security vulnerability.

In a development environment, you might have come across an error when you didn't set `ALLOWED_HOSTS` and tried to access your application from a different machine or domain. For production, we need to be very specific about which hosts are allowed to host our application.

Step 1: Identify your AWS EC2 Public IP and Domain (if available)

When you create an AWS EC2 instance, AWS provides you with a Public IPv4 address and a Public IPv4 DNS. You'll need these to update the `ALLOWED_HOSTS` setting.

1. Navigate to the AWS Management Console.

2. Under the 'Services' dropdown, select 'EC2'.

3. In the EC2 Dashboard, under 'Instances', click on 'Instances' to view a list of your instances.

4. Select the instance you are using for your Django deployment. In the 'Description' tab below, note down the 'IPv4 Public IP' and 'Public IPv4 DNS'.

Step 2: Update the `ALLOWED_HOSTS` in settings.py

Open your Django project in Atom or your preferred code editor. Navigate to the `settings.py` file which is typically in the main application directory.

Locate the `ALLOWED_HOSTS` setting. By default, it's an empty list:

"`python

ALLOWED_HOSTS = []

“`

Update it to include the IP and DNS from your AWS EC2 instance:

“`python

ALLOWED_HOSTS = [‘your-ec2-ipv4-public-ip’, ‘your-ec2-public-ipv4-dns’]

“`

Replace `‘your-ec2-ipv4-public-ip’` and `‘your-ec2-public-ipv4-dns’` with the values you noted down in the previous step.

Note: If you have a domain name pointing to this server, you should add that domain to the list as well.

Step 3: Commit Changes

1. Using your terminal or command prompt, navigate to your project directory.

2. Use the following commands to commit your changes:

“`bash

git add .

git commit -m “Updated ALLOWED_HOSTS for AWS deployment.”

“`

Step 4: Deploy Changes to AWS

1. Push your code changes to your repository.

“`bash

git push origin master

“`

2. SSH into your AWS EC2 instance:

“`bash

ssh -i path-to-your-aws-key-pair.pem ec2-user@your-ec2-ipv4-public-ip

“`

Replace `path-to-your-aws-key-pair.pem` with the path to your AWS key pair and `your-ec2-ipv4-public-ip` with your EC2 instance’s public IP.

3. Navigate to the directory where your Django project is located and pull the latest changes:

“`bash

cd path-to-your-django-project

git pull origin master

“`

4. Restart your server (this could be Gunicorn, uWSGI, etc.) to apply the changes.

Conclusion:

With `ALLOWED_HOSTS` properly set, you’ve taken a crucial step in ensuring your Django application’s security. It’s essential to update this list whenever you’re adding a new domain or IP that should be allowed to host your application. Always be cautious and only allow trusted domains or IPs to ensure the security of your application.

~ Conclusion

As we close the final chapter of our journey into building a REST API with Python & Django, it’s important to take a moment to reflect on the skills and knowledge you’ve acquired, and the powerful capabilities you now possess as a backend developer.

Looking Back: From Basics to Mastery

Starting with the fundamentals, you learned about the importance of backend REST APIs in today’s technology

landscape. From the likes of Facebook and Instagram to small startups, a strong backend is the foundation for many successful digital platforms. As you progressed through this course, you delved into setting up your development environment, mastering Django and Django REST Framework, understanding databases, and much more.

Key Takeaways

1. Development Environment: By setting up tools like Vagrant, VirtualBox, Atom, and ModHeaders, you've established a solid foundation that is not only useful for this project but also for countless future developments.

2. Django Mastery: Django, coupled with Django REST Framework, offers a powerful toolset to build robust and scalable web applications. Through this course, you've gone from creating a simple "Hello World" script to deploying an entire API to AWS.

3. Database Understanding: The core of any backend system is its database. Your deep dive into Django models and understanding of databases will serve as a pivotal knowledge point in your career.

4. APIs & Endpoints: With a hands-on approach, you've created, tested, and iterated on multiple API views and viewsets. You've tackled user authentication, profile management, and feed features – all essential components of modern web apps.

5. Deployment: One of the significant steps of software development is deploying the application. By taking your API live on AWS, you've crossed a milestone that transforms a developer from a hobbyist to a professional.

Looking Ahead: The Future of Backend Development

While you've achieved much, the world of backend development is vast and constantly evolving. Technologies will change, and new methodologies will emerge. It's essential to remain curious and continually seek out knowledge. Consider exploring further into:

- Containers & Microservices: Tools like Docker can be the next step in your journey.

- Database Scaling: Dive deeper into databases, exploring NoSQL options, and understanding scaling strategies.

- Advanced Security: As you build more complex applications, understanding cybersecurity threats and mitigation will become crucial.

Final Words

This course was designed to provide a foundational understanding of building a REST API using Django and Python. You're now well-equipped to design, develop, and deploy backend systems, making your app ideas a reality or collaborating effectively with frontend teams. Remember that learning is a continuous journey. The tech landscape is dynamic, and as a developer, your adaptability and curiosity will define your success.

In the end, whether you're a seasoned developer or just starting out, remember to always remain passionate, keep building, and most importantly, never stop learning. The digital world awaits your next innovation.