



ULTIMATE

# Web API Development with **Django REST** Framework

Build Robust and Secure Web APIs with  
Django REST Framework Using Test-Driven  
Development for Data Analysis and  
Management

An abstract, colorful graphic that occupies the bottom half of the cover. It features swirling, liquid-like patterns in shades of blue, purple, orange, and red, with small, glowing circular elements scattered throughout, resembling a cosmic or molecular structure.

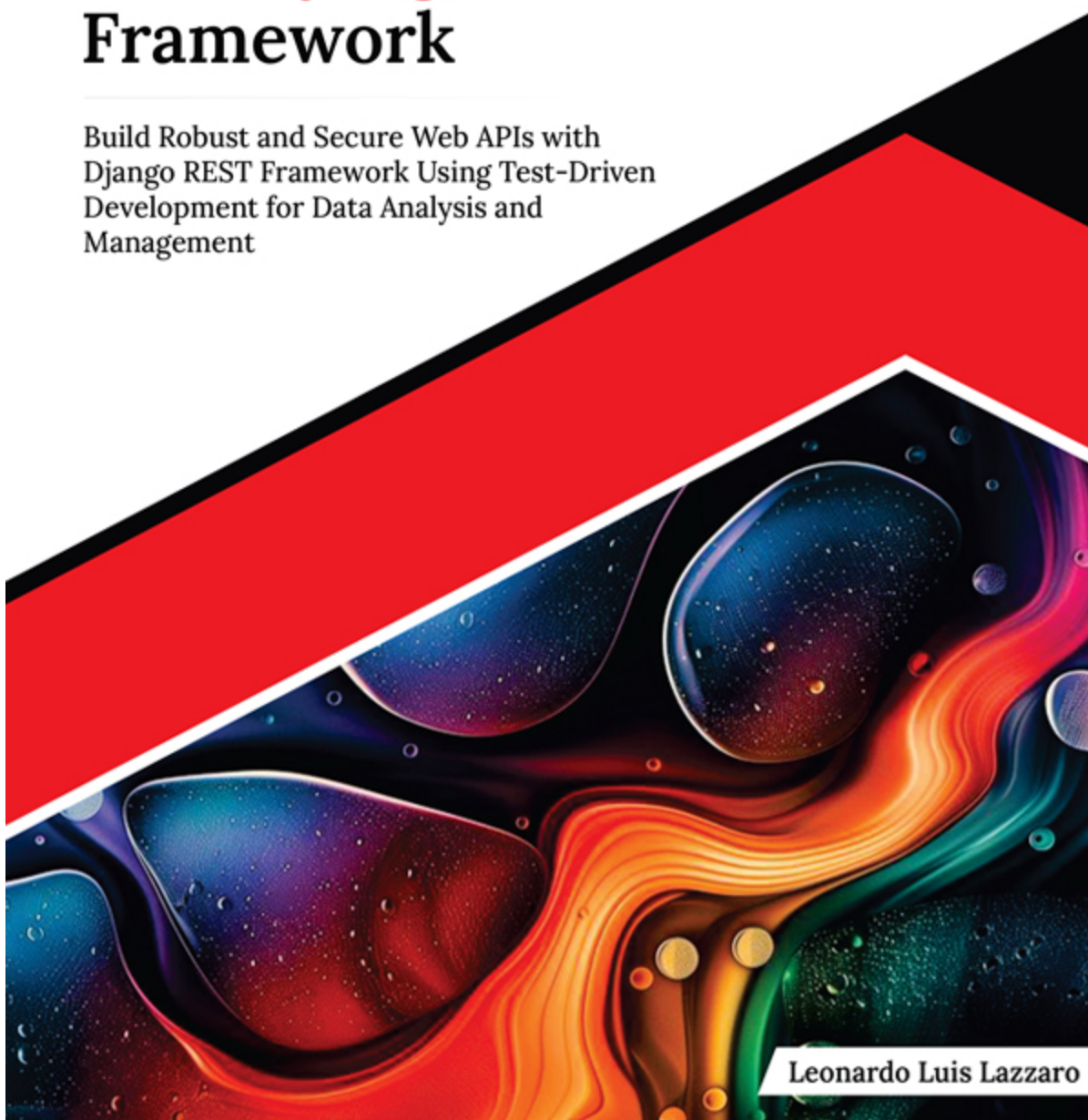
Leonardo Luis Lazzaro



ULTIMATE

# Web API Development with **Django REST** Framework

Build Robust and Secure Web APIs with  
Django REST Framework Using Test-Driven  
Development for Data Analysis and  
Management



Leonardo Luis Lazzaro

# Ultimate Web API Development with Django REST Framework

---

*Build Robust and Secure Web APIs with  
Django  
REST Framework Using Test-Driven  
Development for Data Analysis  
and Management*

---

**Leonardo Luis Lazzaro**



[www.orangeava.com](http://www.orangeava.com)

[OceanofPDF.com](http://OceanofPDF.com)

Copyright © 2025 Orange Education Pvt Ltd, AVA®

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor **Orange Education Pvt Ltd** or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

**Orange Education Pvt Ltd** has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, **Orange Education Pvt Ltd** cannot guarantee the accuracy of this information. The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

**First Published:** January 2025

**Published by:** Orange Education Pvt Ltd, AVA®

**Address:** 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,  
N1 7AA, United Kingdom

**ISBN (PBK):** 978-93-48107-91-6

**ISBN (E-BOOK):** 978-93-48107-26-8

**Scan the QR code to explore our entire catalogue**



[www.orangeava.com](http://www.orangeava.com)

[OceanofPDF.com](http://OceanofPDF.com)

# Dedicated To

*Facultad de Ciencias Exactas at the Universidad de Buenos Aires (UBA)*  
*For The Good Memories and The Opportunities to Build an Unimaginable*  
*Future*

[OceanofPDF.com](http://OceanofPDF.com)



# About the Author

Born in Buenos Aires (la Ciudad de la Furia), Argentina, **Leonardo Luis Lazzaro** has always been fascinated by the idea of building something out of nothing. His first contact with computers began early, fueled by classic video games such as *Maniac Mansion* and *Monkey Island*. At just 12 years old, Leonardo was running his own Bulletin Board System (BBS) using ProBoardBBS Software, making him one of the youngest members of these online communities in Argentina. The BBS allowed him to meet other tech enthusiasts who introduced him to the world of programming. His fascination with computer demos from the demoscene became a major source of motivation for his continued discovery in programming.

Leonardo's academic path led him to study computer science at the prestigious Facultad de Ciencias Exactas, Universidad de Buenos Aires (UBA). He then embarked on a Ph.D. in drug discovery, trying to apply his computational skills to solve complex challenges through GPU simulations. However, his journey took a turn, leading him away from the academia and he became a Ph.D. dropout.

With 12 years of experience in Python, Leonardo has developed a profound expertise in this programming language. He is proficient in several Python frameworks, including Flask, Pyramid, Django, and FastAPI, among others, showcasing his versatility and deep understanding of web development and application design.

[OceanofPDF.com](https://oceanofpdf.com)

# About the Technical Reviewer

**David Wobrock** is a seasoned software engineer with extensive experience in backend web development, cybersecurity, and developer experience. As an active contributor to Django, he plays a significant role in the Django *Triage & Review* team, showcasing his commitment to the advancement of the framework. His primary focus within Django revolves around contributions to the Django ORM and database migrations. Additionally, David is dedicated to maintaining open-source Python packages within the Django ecosystem.

He has worked for several startups, contributing not only to the growth of their technical stacks by developing reliable and secure software but also enhancing team efficiency by providing internal tools, guidelines, and best practices within organizations. He believes that having the right tools—ones that make it easy for developers to do the right thing— is essential for building a great developer experience. Thus, when these tools enable teams, they not only become more efficient but also build more reliable, scalable, and secure products.

[OceanofPDF.com](https://oceanofPDF.com)

# Acknowledgements

I would like to thank the vibrant Python and Django communities for inspiring countless developers to innovate and share knowledge. I also want to thank my colleagues and peers in the tech industry for providing valuable feedback, which has enabled me to refine my understanding of API development.

To the learners and professionals who continue to explore the potential of Django REST Framework and Python—you are the reason this book exists. I hope it serves as a stepping stone for your projects and ambitions. Lastly, a big thank you to my readers for trusting my guidance on this exciting path of API development.

[OceanofPDF.com](http://OceanofPDF.com)



# Preface

The growing need for effective communication between applications and the importance of data-driven decision-making has made APIs a fundamental part of modern technology. *Ultimate Web API Development with Django REST Framework* explores the core principles of building robust and scalable APIs, with a strong emphasis on the Test-Driven Development (TDD) approach to create reliable, maintainable, and high-quality software.

This book is thoughtfully structured to guide readers through mastering the Django REST Framework. It introduces Python enthusiasts to the foundations of web APIs and illustrates their practical use in real-world scenarios. By the time you finish, you will have a thorough understanding of API architecture, enhanced Python programming skills, and hands-on experience developing a recommendation system.

Adapted for backend and full-stack developers, as well as data scientists and engineers, this book bridges the gap between web development and data science. It is especially valuable for those with experience in one field but limited exposure to the other, helping readers broaden their expertise and build confidence in both areas. Covering everything from basic concepts to advanced techniques in optimization and deployment, it equips you with the knowledge and tools needed to succeed in today's tech-driven world.

Whether you are new to API development or an experienced developer aiming to expand your skills, this book delivers practical strategies and actionable insights to elevate your expertise with the Django REST Framework.

**Chapter 1. Django REST and TDD Essentials:** This chapter introduces the fundamentals of Django REST Framework and Test-Driven Development (TDD), covering models, serializers, and views, along with writing tests using Pytest. Readers will set up a foundational project to apply TDD concepts.

**Chapter 2. Building and Testing Basic APIs:** This chapter focuses on designing and implementing simple APIs with Django REST Framework, emphasizing CRUD operations, serializers, API views, and refactoring through TDD practices.

**Chapter 3. Data Models and Processing in Data Science:** In this chapter, readers will learn how to design data models for data science projects, build APIs for data ingestion, and process and transform data using Django integrated with data science libraries.

**Chapter 4. Asynchronous Tasks and Data Processing:** This chapter covers implementing background tasks with Celery, testing data flows, and handling large data sets for scalable data processing.

**Chapter 5. Securing and Scaling Data Science APIs:** In this chapter, readers will learn to secure APIs using custom authentication and permissions, implement strategies for scaling APIs, and ensure data security and privacy when handling sensitive or high-volume data.

**Chapter 6. Developing a Data Science Project:** This chapter provides a comprehensive case study on building a recommendation engine, showcasing the integration of complex data structures, algorithms, and API endpoints in the Django REST Framework.

**Chapter 7. Documenting and Optimizing Your API:** In this chapter, readers will explore techniques for creating comprehensive API documentation, tools for documentation, and methods to optimize API performance for efficiency and scalability.

**Chapter 8. Deploying Your Data Science API:** This chapter covers the steps to prepare a data science API for production, including deployment strategies, best practices for maintaining APIs, and monitoring them in live environments.

**Chapter 9. Final Thoughts and Future Directions:** The concluding chapter summarizes the book's key lessons, highlights best practices in API development and data science, and explores future trends in API and data science integration.

## Downloading the code bundles and colored images

Please follow the link or scan the QR code to download the *Code Bundles and Images* of the book:

<https://github.com/ava-orange-education/Ultimate-Web-API-Development-with-Django-REST-Framework>



The code bundles and images of the book are also hosted on <https://rebrand.ly/2e5d86>



In case there's an update to the code, it will be updated on the existing GitHub repository.

## Errata

We take immense pride in our work at **Orange Education Pvt Ltd** and follow best practices to ensure the accuracy of our content to provide an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

[errata@orangeava.com](mailto:errata@orangeava.com)

Your support, suggestions, and feedback are highly appreciated.

[OceanofPDF.com](http://OceanofPDF.com)

## DID YOU KNOW

Did you know that Orange Education Pvt Ltd offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.orangeava.com](http://www.orangeava.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: [info@orangeava.com](mailto:info@orangeava.com) for more details.

At [www.orangeava.com](http://www.orangeava.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on AVA® Books and eBooks.

## PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [info@orangeava.com](mailto:info@orangeava.com) with a link to the material.

## ARE YOU INTERESTED IN AUTHORIZING WITH US?

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please write to us at [business@orangeava.com](mailto:business@orangeava.com). We are on a journey to help developers and tech professionals to gain insights on the present technological advancements and innovations happening across the globe and build a community that believes Knowledge is best acquired by sharing and learning with others. Please reach out to us to learn what our audience demands and how you can be part of this educational reform. We also welcome ideas from tech experts and help them build learning and development content for their domains.

## REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers

can then see and use your unbiased opinion to make purchase decisions. We at Orange Education would love to know what you think about our products, and our authors can learn from your feedback. Thank you!

For more information about Orange Education, please visit [www.orangeava.com](http://www.orangeava.com).

[OceanofPDF.com](http://OceanofPDF.com)



# Table of Contents

## **1. Django REST and TDD Essentials**

Introduction

Structure

Introduction to Python

*Understanding Variables as References*

*F-Strings*

*Data Structures*

*Tuple*

*Lists*

*Set*

*Dictionary*

*Functions*

*Classes and Objects*

*Error Handling*

*Duck Typing*

*Polymorphism*

*Generators*

*Decorators*

*Standard Modules*

*Setting Up the Python Environment*

*Linting*

*Type Hints*

Understanding HTTP

*Restful API Principles*

*OpenAPI*

Introduction to Django

TDD Concepts and Workflow in Django

Building Basic Models in Django

Writing Tests Using Pytest

*Introduction to Git*

*Project Introduction*

Conclusion

Questions

[Exercises](#)

## **[2. Building and Testing Basic APIs](#)**

[Introduction](#)

[Structure](#)

[API-First Design Approach](#)

[Designing Simple APIs with Django REST Framework](#)

[Testing API Views](#)

[Writing and Testing Serializers](#)

[Developing CRUD Operations](#)

[Refactoring Code with TDD](#)

[Conclusion](#)

[Questions](#)

[Exercises](#)

## **[3. Data Models and Processing in Data Science](#)**

[Introduction](#)

[Structure](#)

[Movie Recommendation System](#)

[Designing Data Models for Data Science Projects](#)

[Extending the Models](#)

[API Architecture](#)

[Creating Endpoints for the User Profile](#)

[Understanding SPARQL](#)

[Getting the Data for Our Project](#)

[Creating and Testing APIs for Data Ingestion](#)

[Conclusion](#)

[Questions](#)

[Exercises](#)

## **[4. Asynchronous Tasks and Data Processing](#)**

[Introduction](#)

[Structure](#)

[Introduction to Background Task Processing](#)

[Introduction to Celery](#)

[\*Celery Local Development Environment\*](#)

[\*Example Task with Celery\*](#)

[\*Background Tasks with Celery and Django\*](#)  
[\*Testing Asynchronous Data Processing\*](#)  
[\*Refactor API for Background Task Processing\*](#)  
[\*Using Cloud File Storage\*](#)  
[\*Handling Large Data Sets and Long-Running Processes\*](#)  
[\*Conclusion\*](#)  
[\*Questions\*](#)  
[\*Exercises\*](#)

## **5. Securing and Scaling Data Science APIs**

[\*Introduction\*](#)  
[\*Structure\*](#)  
[\*Understanding Django's Authentication System\*](#)  
[\*Introduction to Django's Middleware\*](#)  
[\*Understanding Django Middleware\*](#)  
[\*Setting Up the Django Admin\*](#)  
[\*Groups and Permissions\*](#)  
[\*Implementing Authentication Methods\*](#)  
[\*Basic Authentication\*](#)  
[\*Token Authentication\*](#)  
[\*JWT \(JSON Web Token\) Authentication\*](#)  
[\*Custom Permissions for Sensitive Data\*](#)  
[\*Ensuring Security in Data APIs\*](#)  
[\*Enforce HTTPS\*](#)  
[\*Use Secure Authentication Methods\*](#)  
[\*Implement Rate Limiting\*](#)  
[\*Validate Input Data\*](#)  
[\*Implement Proper CORS Handling\*](#)  
[\*Limit Data Exposure\*](#)  
[\*Log and Monitor API Usage\*](#)  
[\*Use Strong API Keys and Secrets Management\*](#)  
[\*Regularly Update Dependencies\*](#)  
[\*Conduct Security Audits and Penetration Testing\*](#)  
[\*Conclusion\*](#)  
[\*Questions\*](#)  
[\*Exercises\*](#)

## **6. Developing a Data Science Project**

[Introduction](#)

[Structure](#)

[Introduction to Recommendation Engines](#)

[The Recommendation Algorithm](#)

[The Role of Cosine Similarity](#)

[Preparing the Local Environment](#)

[Cleanup the Data](#)

[Recommendation System Implementation Using Django](#)

[Validating Recommendation System with Unit Tests](#)

[Recommendation System Implementation](#)

[Conclusion](#)

[Questions](#)

[Exercises](#)

## **7. Documenting and Optimizing Your API**

[Introduction](#)

[Structure](#)

[Introduction to OpenAPI](#)

[\*Understanding OpenAPI\*](#)

[\*Using OpenAPI\*](#)

[\*Setting Up OpenAPI with drf-spectacular\*](#)

[\*Documenting API Endpoints\*](#)

[Performance Optimization Techniques for APIs](#)

[\*Data Caching Techniques\*](#)

[\*Reducing Payload Size\*](#)

[\*Rate Limiting and Throttling\*](#)

[Conclusion](#)

[Questions](#)

[Exercises](#)

## **8. Deploying Your Data Science API**

[Introduction](#)

[Structure](#)

[Introduction to Gunicorn](#)

[Configuring Gunicorn for Django Deployment](#)

[Understanding and Creating Dockerfiles for Django](#)

[Using the Image Registry](#)  
[Introduction to Kubernetes](#)

[Cluster](#)

[Node](#)

[Scheduler](#)

[Pods](#)

[Deployments](#)

[ReplicaSets](#)

[Services](#)

[ConfigMaps and Secrets](#)

[Ingress](#)

[StatefulSets](#)

[Configuring a Kubernetes Cluster for a Django Application](#)

[Adding Liveness and Readiness Probes](#)

[Conclusion](#)

[Questions](#)

## **[9. Final Thoughts and Future Directions](#)**

[Introduction](#)

[Structure](#)

[Key Takeaways from the Book](#)

[Best Practices in API Development and Data Science](#)

[Future Trends in Web APIs and Data Science](#)

[Final Thoughts and Encouragement for Continuous Learning](#)

[Conclusion](#)

## **[Index](#)**

[OceanofPDF.com](#)

# **CHAPTER 1**

## **Django REST and TDD Essentials**

### **Introduction**

This chapter introduces key concepts for modern API development with Python and Django Rest Framework. You will learn the principles of HTTP and the basics of Django Rest Framework. We will explore the dynamic and strongly typed nature of Python, how to handle web requests with HTTP, and Django’s “batteries-included” approach for building web applications. By showing you the importance of Test-Driven Development (TDD) and version control with Git, we will guide you through developing robust and maintainable code. Additionally, we will touch on Python’s standard library and Django’s project setup. This chapter sets the stage for understanding essential web development concepts and practices.

### **Structure**

In this chapter, we will discuss the following topics:

- Introduction to Python
- What is HTTP?
- Introduction to Django
- TDD Concepts and Workflow in Django
- Building Basic Models in Django
- Writing Tests Using Pytests
- Introduction to Git
- Project Introduction

### **Introduction to Python**

Python is a high-level interpreted programming language. Due to its high level, it allows the development of complex applications with fewer lines of



code compared to many other languages.

Unlike many languages that use semicolons or braces to mark the end of a statement, Python relies on newlines and tabulations.

```
# Initialize counter
count = 1

# Loop from 1 to 10
while count <= 10:
    # Check if the number is even
    if count % 2 == 0:
        print(f"{count} is even")
    else:
        print(f"{count} is odd")
    # Increment the counter
    count += 1
```

The provided code demonstrates a loop that executes as long as the variable `count` remains less than or equal to 10. Within the loop, we check if the count is even by using the modulus operator `%`. If the count is divisible by 2 (`count % 2 == 0`), it prints that the number is even. Otherwise, it prints that `x` is odd. After each iteration, `x` is incremented by 1 with `x += 1`.

Indentation can be seen in the loop when everything runs at the same level. The same applies to the conditionals `if` and `else`. The code inside the `while` will be executed until the condition is false.

Python is a strongly typed language, enforcing strict type constraints without implicit conversion between types. It is dynamically typed, meaning it determines variable types at runtime instead of statically typed languages that perform type checking at compile time. Let us see what strongly typed and dynamically typed means with examples:

```
def add(a, b):
    return a + b

# Calling add with integers
result1 = add(5, 7)
print(result1) # Output: 12

# Calling add with strings
result2 = add("Hello, ", "World!")
print(result2) # Output: Hello, World!
```

```
# Attempting to add incompatible types
result3 = add("Number: ", 5)
print(result3) # This will raise a TypeError at runtime
```

In the preceding code, the function `add` uses the binary operator `+` against the parameters `a` and `b`. When we call the function `add` with parameters of the same type, the function will return the result of the operation. However, a **TypeError** will be raised at runtime when the types differ. Type compatibility is only checked at runtime.

Being strongly typed means it fails on implicit type conversions. Let us see an example to understand what it means:

```
a = 10 # Integer type
b = "5" # String type
# Attempting an arithmetic operation between an integer and a
string
result = a + b
```

In this code snippet, we attempt to add an integer (`a`) to a string (`b`). Because Python is a strongly typed language, it does not implicitly convert the string `b` into an integer to perform the addition. Consequently, this code will raise a **TypeError** at runtime, indicating that you cannot directly add an `int` (integer) and `str` (string) due to their incompatible types.

The categorization of programming languages is often based on their typing discipline (static versus dynamic) and type strength (strong versus weak). The following table outlines how various programming languages fall into these categories:

	Statically Typed	Dynamically Typed
Strongly Typed	Java, C#	Python, Ruby
Weakly Typed	C, C++	JavaScript, PHP

*Table 1.1: Categorization of programming languages based on two different typing characteristics*

## Understanding Variables as References

Python differs from other programming languages; it doesn't need to declare variable types beforehand. In Python, variables act as pointers to objects, not the objects themselves.

To understand how it works, check this simple Python code:

```
x=5
y=x # At this point, x and y point to the same object 5.
x=10 # Now, x points to a different object, 10, but y points
to 5.
```

## F-Strings

The Python Enhancement Proposal (PEP) 498 (<https://peps.python.org/pep-0498/>) proposed adding f-strings to the Python language and it was introduced in Python 3.6. F-string is a way to include the value of Python expressions inside string literals. F-strings are denoted by an f or F prefix before the opening quotation mark of a string literal.

```
name = "Pepe"
age = 23
message = f"Hello, {name}. You are {age} years old."
print(message)
```

The output will be:

```
>>> Hello, Pepe. You are 23 years old.
```

F-string also supports formatting options when using the curly braces:

```
number = 3.13
formatted_number = f"{number:08.2f}"
print(formatted_number)
```

The preceding f-string specifies options after the colon “:”. The **08** specifies that the string should be at least eight characters and the **.2f** dictates that the number will be formatted with two decimal places. The string will be padded with zeros:

```
>>> 00003.13
```

There are plenty of options when using the f-string. Make sure to check Python documentation at:

<https://docs.python.org/3/library/string.html#formatspec>

## Data Structures

Python has a rich set of built-in data structures, allowing you to store and access data. Any Python programmer needs to know and understand these data structures.

## Tuple

A tuple in Python is an ordered collection of immutable objects of fixed sizes. Once a tuple is created, changing its size is impossible and does not support item assignment.

```
# Let's create a new tuple
tuple_1 = (1,2,3)
# Access the element at position 0
tuple_1[0]
>> 1
```

## Lists

A list is an ordered collection of objects that does not have a fixed length, and it supports item assignment.

```
# Let's create a new list
example = [1,2,3]
# Access the element at position 0
example[0]
>>> 1
# change the element in position 0
example[0] = -1
# You can iterate through a list
for index, element in enumerate(example):
    print(f"element {element} at index {index}")
```

The form iterates through the elements of the list. Enumerate is a Python built-in function that adds a counter to an iterable and returns it as an enumerate object.

## Set

A Set is an unordered collection of unique objects with no fixed size. Since it is unordered, it is not subscriptable.

```
# Let's create a new set
```

```

example = {1,2,3}
# print the size of the set
print(len(example))
>>> 3
# We can check if an element exists using the in-operator
1 in example
>>> True
# if we add an existing element, its size does not change
example.add(1)
len(example)
>>> 3
# We can do a set difference
example - {1}
>>> {2, 3}
# or intersection
example.intersection({1})
>>> {1}

```

## Dictionary

Dictionaries are key-value stores that are indexed by keys and are unordered. In Python, it is widespread to work frequently with dictionaries.

```

# Let's create a dictionary
example = {"key_1": 1, "key_2": "value" }
# It is possible to access the value using a valid key
example["key_1"]
>>> 1
# The get operator gets the key or the default value
example.get("key_3", None)
>>> None
# It is possible to iterate a dictionary
for key, value in example.items():
    print(f"key {key} value {value}")

```

## Functions

Python uses indentation to define blocks of code:

```
def hello_world(name: str, greeting="Hello") -> None:
    print(f"{greeting} world! {name}")
for i in range(1, 100):
    print(i)
```

Functions are defined with a **def** keyword, and support positional and keyword arguments, as shown in the function **hello\_world**.

In programming, there are two ways to pass arguments to a function when it is called: pass by reference or pass by value. When a function argument is passed by reference, the function receives a reference to the original variable rather than a copy of its value.

Pass by copy (value) means a function receives a copy of the argument's value. Changes to this copy do not affect the original variable.

Python employs a mechanism often described as 'pass-by-object-reference' or 'pass-by-assignment'. The function receives a reference to the object and is not copied. How the function interacts with the passed object depends on whether the object is mutable or immutable.

Let us see an example with immutable objects:

```
from typing import Any
def modify(x: Any) -> Any:
    x = x + 1
    return x

a = 5
print(modify(a)) # Prints: 6
print(a)         # Prints: 5, showing 'a' is unchanged outside
the function
```

Numbers, strings, and tuples are examples of immutable objects. When you pass an immutable object to a function and that function attempts to modify it, a new object is created and assigned within the function's local scope.

With mutable objects, the behavior is different, and the object is changed, producing a side effect:

```
def modify(lst: list) -> None:
    lst.append(4) # Modifies the list in-place

my_list = [1, 2, 3]
modify(my_list)
```



```
print(my_list) # Prints: [1, 2, 3, 4], showing 'my_list' was
modified by the function
```

As a good practice, you should avoid side effects and your functions should not modify the mutable objects passed as arguments.

## Classes and Objects

A class in Python is a blueprint for instantiating objects, which has a set of attributes and methods. An object is an instance of a class. Each object can share a common behavior through methods defined in the class.

```
class SpruceMoose:
    def __init__(self, name):
        self.name = name
    def quack(self):
        print("Quack!")

class Duck:
    def quack(self):
        print("Quack Quack!")
```

In the preceding code, we can see two Python classes. The **SpruceMoose** has two methods implemented: the **\_\_init\_\_** and the **quack**. Methods with double underscore are known as “dunder” or “magic” methods. Dunder methods allow your classes to implement and interact with built-in Python operations. They allow you to define how your objects should behave in various contexts, such as when they are being added together, represented as strings, or compared.

```
# Creating instances of the classes
moose = SpruceMoose("Spruce")
duck = Duck()

# Calling the quack method on each instance
moose.quack() # This will print "Quack!"
duck.quack()  # This will print "Quack Quack!"
```

The code prints the **"Quack!"** and **"Quack Quack!"** respectively.

## Error Handling

An exception is an unexpected event that disrupts the normal flow of a program. When an exception occurs, Python creates an Exception object that will propagate up the call stack. Handling exceptions is crucial for writing robust code that can deal with unexpected errors gracefully. Python uses a try-except block to catch exceptions. Let us see an example when Python code tries to access a key that does not exist in a dictionary:

```
def main():
    traffic_light = {"red": 2, "green": 0, "yellow": 1}
    traffic_light["blue"]
main()
print("finish")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 3, in main
KeyError: 'blue'
```

If we want our program to print “finish”, we need to handle the **KeyError** exception properly:

```
>>> try:
...     main()
... except KeyError as ex:
...     print(f"Key {ex} does not exists")
...     print("finish")
>>> Key 'blue' does not exist
>>> finish
```

The new code with the try-except catches the exception and prints that the “blue” key does not exist. Finally, it prints “finish”. There are situations when the function body we call is more complex and could return another type of exception. In this case, our try-except will only catch the **KeyError**. In this situation, it is tempting for developers to catch the generic exception, and by doing this, it will catch all possible exceptions. It is a good practice to catch specific exceptions, not generic ones since it could catch unexpected errors and obscure the program’s logic.

Python try-except also supports else and finally clauses. Let us see an example:

```
def divide_function(n: int) -> None:
```

```

try:
    result = 0 / n
except ZeroDivisionError:
    print("Divided by zero.")
else:
    print("Division successful.")
finally:
    print("Always printed.")

```

The **divide\_function** accepts an integer "n", which is used as a divisor in the division. Then n equals zero, it will print "**Divided by zero.**" and "**Always printed.**". When n is not equal to zero, it will print "**Division successful.**" and "**Always printed.**". The else branch is only executed when there is no exception and the finally is always executed. Usually, the final branch is used to clean up resources or close connections.

## [Duck Typing](#)

Duck typing is a term used in dynamic languages that do not have robust type systems. The idea comes from the phrase, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." Since our two classes implement the quack method, instances of both **SpruceMoose** and **Duck** fulfill an interface and allow objects to be interchanged regardless of their type.

## [Polymorphism](#)

Polymorphism is the ability of different objects to respond to the same method call. Duck typing can be seen as a polymorphism, where an object's compatibility is determined at runtime by the methods it implements, not by its inheritance hierarchy.

Python supports multiple class inheritance, allowing polymorphism.

```

class Airplane:
    def __init__(self, name):
        self.name = name
    def horn(self):
        raise Exception("Airplanes do not need a horn!")

```

```
class SpruceMoose(Airplane):  
    def horn(self):  
        print("Honk Beep hoonk!")
```

The preceding code shows an example of class inheritance. The horn method is overridden in the **SpruceMoose** class.

## Generators

Simple Generators were introduced in the PEP255 (<https://peps.python.org/pep-0255/>). The PEP was accepted and introduced in Python 2.2. Generators simplify the creation of iterators. The PEP255 also introduced the yield statement to Python, which allows the suspension and resumption of the execution of a function. Generators are defined as a function, but instead of returning a value with a return, it yields a sequence of values using the yield.

```
from typing import Generator  
  
def fibonacci() -> Generator[int, None, None]:  
    current, next_value = 0, 1  
    while True:  
        yield current  
        current, next_value = next_value, current + next_value
```

The **fibonacci** function generates an infinite sequence of Fibonacci numbers using a generator. It efficiently produces values on-demand without precomputing or storing the entire sequence by continuously yielding the following number. This approach showcases the power and efficiency of Python's generators for handling infinite sequences in a resource-friendly manner.

Here is an example of how to use the **fibonacci** to generate the first 10 numbers:

```
for index, fib_number in enumerate(fibonacci()):  
    print(fib_number)  
    if index > 10:  
        break
```

The preceding code iterates the Fibonacci numbers using the enumerate. Enumerate is a built-in Python function that adds a counter to an iterable,

enabling you to loop over something and have an automatic counter. The loop body prints the Fibonacci number and checks if the index is more than 10. When it is bigger, it stops the iteration; otherwise, it will loop indefinitely.

Like list comprehensions, generator expressions allow you to create generators concisely and straightforwardly. They look a lot like list comprehensions but use parentheses instead of square brackets:

```
squares = (x*x for x in range(10))
```

The generator above computes the square of each number from 0 to 9.

Later in the Pytest section, we will see that understanding generators is critical in creating test fixtures.

## Decorators

Decorators were introduced by PEP318 (<https://peps.python.org/pep-0318/>) and have been available since Python 2.4. Decorators allow the modification of functions and methods using other functions. They are denoted by the @ symbol and placed above a function definition. Decorators make the code clean, enabling code reuse.

```
def log_args(func):
    def wrapper(*args, **kwargs):
        print(f"Arguments were: {args}, {kwargs}")
        return func(*args, **kwargs)
    return wrapper
```

The **log\_args** decorator logs the arguments of a function. The wrapper function captures the decorated function's arguments (\*args for positional and \*\*kwargs for keyword arguments) and prints them before calling the original function with those arguments. This pattern allows developers to inject logging or other pre- and post-invocation behavior into any function, improving debugging and monitoring without altering the original function's code.

Here is an example of how to use the **log\_args** decorator:

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

When we call `add`, it will print the arguments:

```
add(10, 2)
Arguments were: (10, 2), {}
```

Later in the `pytest` section, we will often use decorators since most of the `pytest` features are used by decorators.

## Standard Modules

Python's standard library is vast and includes numerous modules. Given its broad scope, covering all of it in an introductory segment is impossible. This section will show the most common modules used while working with a Django project.

Let us start with **`pathlib`**. The **`pathlib`** is an object-oriented module to handle filesystem paths. One of its common uses is in the project settings.

Let us see an example:

```
from pathlib import Path
# BASE_DIR is the project root (the directory containing
# manage.py)
BASE_DIR = Path(__file__).resolve().parent.parent
# This is how you would define the location of the static
# files directory
STATIC_ROOT = BASE_DIR / "staticfiles"
```

As you can see, the `/` is the operator to join paths; the **`pathlib`** module was introduced in Python 3.4, and it offers a great way to handle the filesystem.

JavaScript Object Notation (JSON) is a universally recognized format for storing and transferring data. Serialization refers to converting objects into JSON format, whereas deserialization involves converting JSON back into objects. The standard library has a module that works with JSON, the **`json`** module.

Here are some examples of how to use the JSON module:

```
import json
# let's create a dictionary to represent a person
person = {
    "name": "John",
```



```

"age": 30
}
# serialization of the object
person_json = json.dumps(person)
# deserialization of the json, loads returns a dictionary
person_dict = json.loads(person_json)
assert person == person_dict

```

When the object is not serializable, dumps will raise **TypeError**. To make the object serializable, you must provide a function that translates the object to a dictionary and pass it using the default parameter.

```

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def person_serializer(obj: Any) -> dict:
    if isinstance(obj, Person):
        return {"name": obj.name, "age": obj.age}
    raise TypeError(f"Type {type(obj)} not serializable")

p = Person("John", 30)
import json
json.dumps(p, default=person_serializer)

```

The datetime module is another helpful module to learn. While Django provides several utilities for handling date-times, there might be occasions when it is necessary to resort to the functionalities of the standard datetime module. Let us see an example of the helpful **timedelta** function:

```

from django.utils import timezone
from datetime import timedelta
# with Django it's important to always use datetime timezone
aware
current_datetime = timezone.now()
# let's add 7 days to the current_datetime
future_datetime = current_datetime + timedelta(days=7)

```

Python offers a plethora of other modules to explore. We advise readers to consult the official Python documentation for a deeper understanding of these modules.

## Setting Up the Python Environment

Developers usually work on several projects at once and these projects have their dependencies and different Python versions. The Python eco-system has several tools to manage multiple projects on the same machine. Virtualenv is a tool in Python used to create isolated Python environments. Each virtualenv will have a different set of libraries installed and the developer can easily switch between them.

There are other tools like pyenv and poetry but in this book, we will use a tool that offers a unified experience rye. Rye is a project and package management tool for Python. It will allow us to install different Python versions, have different virtual environments, and manage project dependencies.

First, download **rye** from the official website <https://rye-up.com/>; once you have the binaries downloaded for your operating system, make it executable and move it to your **PATH** directory:

```
gunzip rye-aarch64-macos.gz
chmod +x ./rye-aarch64-macos
move rye-aarch64-macos rye
./rye
```

Now let us install Python 3.12 using rye:

```
rye fetch 3.12
```

The next step is to initialize our project directory. We can do this with rye:

```
mkdir recommendation_system
cd recommendation_system
rye init
```

The command will create a Python project structure with the following directory tree:

```
$ tree -f recommendation_system
recommendation_system
├─ recommendation_system/README.md
├─ recommendation_system/pyproject.toml
└─ recommendation_system/src
    └─ recommendation_system/src/recommendation_system
```

└─

recommendation\_system/src/recommendation\_system/\_\_init\_\_.py

Rye will create an empty **README.md**. This file usually includes a project description and a quick start for developers. Then we have the **pyproject.toml** file, a configuration file for a Python project. This file was introduced in PEP518 (<https://peps.python.org/pep-0518/>). The file **pyproject.toml** also contains settings for dependency management, package metadata, tool configurations, and more. The file objective is to standardize configuration settings and stages of project development. Finally, we have our src directory, where we will put the source code of our Django project. The **\_\_init\_\_.py** file in Python marks directories on disk as Python package directories.

Now we will create the virtualenv:

```
rye sync
```

For adding dependencies, you will need to use rye:

```
rye add django
```

```
rye sync
```

## Linting

Linters are tools that analyze the source code to detect errors, bugs, or style errors. Using a linter with Python is very important due to the dynamic typing, meaning that types are checked at runtime. Linters such as Mypy can perform static type checking to catch these errors early. Linters also ensure that code adheres to style guides like PEP 8, improving readability and maintainability.

You can install linter as a development dependency using the flag **--dev**:

```
rye add --dev ruff mypy
```

```
rye sync
```

```
rye run black
```

Every time we add a new dependency, we need to sync. Rye allows the execution of black using the run command, as shown.

## Type Hints

Type hints were introduced in PEP 484 (<https://peps.python.org/pep-0484/>) and became part of Python 3.5. Type hints or annotations are a way to explicitly specify the type of variables, function parameters and return types. Due to the nature of dynamically typed languages like Python, type annotations help developers unfamiliar with the project since they are also tools for documents. Type annotation can also be used to do static checks. For example, tools such as Mypy can analyze the code for type consistency without executing it.

Let us see some examples of how to use type hints with Python:

```
age: int = 25
name: str = "Pepe"
```

The preceding example shows how to annotate variables with their expected type.

```
numbers: list[int] = [1, 2, 3]
student_scores: dict[str, int] = {"Alice": 90, "Bob": 95}
```

It is also possible to annotate collections, as shown in the previous code snippet.

```
def greet(name: str) -> str:
    return f"Hello, {name}"
```

Functions can have their parameters and return values annotated, as shown in the greet function.

## Understanding HTTP

The Hypertext Transfer Protocol (HTTP) is a request-response protocol used for communication between a client and a server. When a client requests access to a specific resource via a URL, the server processes this request and returns a response with the content. A URI is a string of characters to identify a name or a resource. A URL is a specific type of URI that identifies a resource and provides the schema to access it (HTTP, HTTPS, and more). HTTP protocol uses methods for interacting with the server and responses have status codes. Knowing how to use the HTTP protocol is essential when building an API. It is stateless, meaning there is no memory between each request and they are independent.

Let's review the different HTTP methods:

HTTP Method	Description	Idempotent	Safe	Use case
<b>GET</b>	Returns a representation of the resource.	Yes	Yes	Retrieve data without changing it.
<b>POST</b>	Sends data to be processed on the server. Usually, it changes the state of the server.	No	No	Create new instances of an object.
<b>PUT</b>	Sends data to update an existing resource or creates a new one if it does not exist.	Yes	No	Update an object with new data.
<b>DELETE</b>	Deletes the specified resource.	Yes	No	Remove an object.
<b>HEAD</b>	It is the same as GET, but the server only returns the headers. The body is empty.	Yes	Yes	Retrieve metadata without the body.
<b>CONNECT</b>	Establishes a tunnel to the server identified by the target resource.	No	No	Used for SSL tunneling.
<b>OPTIONS</b>	Returns the communication options for the specified resource.	Yes	Yes	Discovering allowed methods on a resource.
<b>TRACE</b>	Performs a message loop-back test along the path to the target resource.	Yes	Yes	A debugging tool to see changes made by intermediaries.
<b>PATCH</b>	Sends partial data to update a resource.	No	No	Partial update of an object.

**Table 1.2: HTTP Methods**

It is a standard practice for servers to return a status code with every response to a client's request. The status code indicates if the request was successful or not. Status codes are categorized into five different classes:

- **1xx:** Informational. The server has received the initial part of a request and is still processing it, but additional steps are required to complete the request.
- **2xx:** Success. Indicates that the request was accepted, processed and accepted by the server.
- **3xx:** Redirection. Indicates that the client needs to perform an additional operation to complete the request, often involving a redirect to another URI.
- **4xx:** Client errors. The request contains invalid syntax and the server rejects it. The client it needs to modify the request before resending it.
- **5xx:** Server errors. The server failed to process the request due to an unexpected error.

**Info:**

***Idempotent** means the method can be called multiple times with the same parameters and should return the same result. Idempotency is a critical property to have when building APIs.*

***Safe** means that the methods should be read-only and not change the server's status.*

*It is important to understand that you can implement a GET endpoint that is not idempotent and safe, but you should never do this. When an API does not stick to the expected behavior, its users will have a hard time, and the API design will be poor.*

Here is a list of relevant status codes when developing an API:

Status Code	Category	Description
200	Success	Ok. The request was processed successfully. Usually returned when doing a GET, PUT or POST.
201	Success	Created. Successfully created the resource. Usually returned when making POST requests.

204	Success	No content. Successfully processed the request. Usually returned when deleting a resource.
301	Redirection	Moved Permanently. Indicates that the resource has been moved to a new URI permanently.
302	Redirection	Found. Temporarily redirects to a different URI, suggesting future requests still use the original URI.
400	Client Error	Bad Request. The request was malformed or invalid, which is common in API calls with incorrect parameters.
401	Client Error	Unauthorized. The request lacks valid credentials, indicating that authentication is required and has failed or not been provided.
403	Client Error	Forbidden. The server understands the request but refuses to process it, often due to insufficient permissions.
404	Client Error	Not Found. The resource was not found.
405	Client Error	Method not allowed. The method used in the request is not allowed for the resource.
409	Client Error	Conflict. The request conflicts with the current state of the server. Usually happens when there is a duplicate entry or editing conflict.
422	Client Error	Unprocessable Entity. The request was well-formed but could not be followed due to semantic errors.
429	Client Error	Too many requests. The client has sent too many requests in a short time.
500	Server Error	Internal Server Error. A generic error message when the server encounters an unexpected condition.
502	Server Error	Bad Gateway. The server acted as a gateway or proxy and received an invalid response from the upstream server.
503	Server Error	Service Unavailable. The server is not ready to handle the request, typically due to maintenance or overloading.
504	Server Error	Gateway Timeout. The server acted as a gateway or proxy and did not receive a timely response from the upstream server.

**Table 1.3:** Most common HTTP Status

HTTP headers describe the properties of data sent in requests or responses, carrying metadata such as the data's format, encoding, length, and instructions for its processing by the client or server.

There exist four types of headers:

- **General:** Not related to the data sent in the body. They can exist in the request or response.
- **Entity:** Has information about the body sent. This type usually refers to the length or MIME type.
- **Request:** These headers include information about the resource to be fetched or the client. Examples are the format for the response or details about the client software.
- **Response:** Additional data about the returned response, like the location or information about the server.

Here is a list of the most common HTTP headers:

Header name	Type	Description
Content-type	Entity	Specifies the format of the data in the request or response body.
Content-Length	Entity	Specifies the size in bytes of the data in the response body.
User-Agent	Request	Contains information about the client software.
Accept	Request	Informs the server of the formats that the client can process
Authorization	Request	Contains a security token used to authenticate against the server.
Cookie	Request	Contains stored HTTP cookies previously sent by the server with the Set-Cookie header.
Location	Response	Used in responses to indicate the URL to redirect a page to or where a new resource was created.
Set-Cookie	Response	Sends a cookie from the server to the client to store and send back in future requests.

*Table 1.4: Most common HTTP headers*

Several tools exist for interactions with API. Curl (client URL) is a command line tool we will use through the book. Let us see some examples:

```
curl -X GET "https://restguide.info.com/api/v1/movies" -H  
"Accept: application/json"
```



The preceding command makes requests to the URL <https://restguide.info.com/api/v1/movies> using the method GET and using the headers Accept: **application/json**.

```
curl -X POST "http://restguide.info.com/ap/v1/movies" \  
  -H "Content-Type: application/json" \  
  -d '{"title": "Inception", "director": "Christopher Nolan",  
    "year": 2010}'
```

The last curl example shows how to execute a request using the POST method and sending a payload with a movie to create. It also uses the header Accept: application/json.

#### **INFO:**

*Several open-source tools exist to design, test, document, and share APIs. Here is a list of popular tools:*

- **Swagger:** *This is a set of tools for designing APIs with the OpenAPI standard. It includes tools to create, document and mock APIs.*
- **Insomnia:** *Open-source API client with cookie management support that can generate code.*
- **qw:** *A tool to create fake APIs according to a specification.*

## **Restful API Principles**

Roy Fielding introduced the concept of Representational State Transfer (REST) in his doctoral dissertation titled “Architectural Styles and the Design of Network-based Software Architectures”. REST is an architectural style for designing network applications.

Roy, in his dissertation, does not strictly tie REST to HTTP. The dissertation describes REST as an architectural style with constraints and principles for designing distributed systems.

We are separating the architectural style from HTTP because adhering to REST’s principles allows the creation of APIs without relying on HTTP. You can implement a REST API with other technologies.

Most real-world RESTful systems are implemented over HTTP, given the alignment between REST principles and HTTP semantics. In the

dissertation, Fielding uses HTTP to illustrate how REST can be applied.

The six principles are as follows:

- **Stateless:** The server should not store anything about the client's state between requests. Each request is independent and must contain all the information the server needs.
- **Client-Server Architecture:** The client and server are different entities. The client is responsible for the user interface, and the server is responsible for the backend and data storage.
- **Layered System:** The API can be composed of multiple layers. The client cannot tell whether it is connected directly to the end server or with intermediary layers.
- **Cacheability:** Responses can be labeled as cacheable. If a response is cacheable, the client cache can reuse it for equivalent responses in the future.
- **Uniform Interface:** All components interact through a consistent interface, simplifying the architecture and improving visibility and portability.
- **Code on Demand (optional):** The server can return code to extend the client functionality. Code on demand could potentially expose a security vulnerability and its use has to be carefully considered.

Common misconceptions about REST include the notion that it requires JSON or is solely intended for CRUD operations. REST is not limited to CRUD. RESTful services can offer other operations beyond basic CRUD, and not every RESTful service needs to implement all CRUD operations. JSON is a popular choice of data interchange due to its lightweight nature and ease of use. RESTful services can return binary data, like MessagePack or Protocol Buffers. Some other services can return CSV, images, or even XML.

The principles are sufficiently generic, allowing you to opt for technology that suits your needs or adapt to emerging technologies. Designing an API involves following these principles and choosing the right technologies to solve the problems. This chapter will focus on JSON and some CRUD operations since we plan to give this API to external users to operate on our project management tool.

## OpenAPI

OpenAPI is an open-source framework that defines a standard to describe the structure and capabilities of Restful APIs. The specification allows developers and systems to understand and interact with the API without direct access to the resource, documentation or code. The OpenAPI Specification includes information about API endpoints, request and response types, authentication methods, and other relevant data for consuming the API effectively. With OpenAPI, it is possible to generate documentation, clients, and test cases. Since it is a standard, it allows for reduced time to integrate APIs by minimizing the need for communication between teams or companies.

## Introduction to Django

Web frameworks allow us to develop web applications, services, and APIs. These frameworks provide an abstraction to focus on the problem we are solving, not on technicalities. Web frameworks provide a standard way to build and deploy projects, which allows a team of people to communicate better.

Here is a simplified explanation of how the web framework works upon receiving a request from a client.

### **Handling Requests:**

Everything starts when the web server receives the HTTP request from the client, which is sent to the framework for processing. Then, the web framework URL routing parses the URL of the incoming request and routes the request to the appropriate controller or view.

**Processing Requests:** Once the framework identifies which controller or view needs to be executed, it executes it using the request as a parameter.

Sometimes, just before calling the view, some frameworks call middleware using the request as a parameter. These middlewares could be used for multiple purposes like security checking or adding information to the context.

The controller or view does an input validation on the request data to ensure the request is valid. The controller or view usually executes some business

logic through an interface if the request has valid data. This business logic uses an Object relationship mapper (ORM) to interact with the database.

### **Generating Response:**

When the view or controller finishes the execution, the framework could call middleware and use the response as a parameter to perform additional operations.

With the result of the controller or view, the framework will generate a response using a template or serializer. Serialization is the process of converting a model into a format like JSON. Once the serialization is done, it will create a response object with the JSON data and other information, like headers and response status.

### **Sending Responses:**

The response object with the data and headers is sent to the web server. The web server will send this response to the client, parse the JSON and use it as required.

Written in Python, Django stands out as a widely used open-source web framework. Adopting a ‘batteries-included’ approach, the framework offers an extensive range of built-in features. Django’s ORM is a powerful feature that abstracts SQL queries into Python code, enabling developers to interact with databases more securely and efficiently, accelerating development and enhancing code quality by minimizing SQL injection vulnerabilities and reducing repetitive code. Django also follows the Don’t Repeat Yourself (DRY) principle, promoting the reusability of components or applications.

#### **INFO:**

*Understanding the difference between project and application is important when working with Django. A Django project is the entire web application and serves as the container for all settings, configurations, and applications. It is essentially the whole website and can contain multiple components or modules, each serving a specific function. Django application is a self-contained package designed to do one thing and do it well, following the Unix philosophy. An application is a reusable module that can perform a specific function and can be included in multiple projects.*

We already installed Django with Rye before. Now, we will structure the Django project using the **src** directory.

First, start a new project using the **django-admin** command:

```
cd recommendation_system
rye run django-admin startproject recommendation_system
```

This will create a **recommendation\_system** directory with the basic Django project structure. Now, we will reorganize using the **src** directory. First, move the **manage.py** file:

```
mv recommendation_system/manage.py src/
```

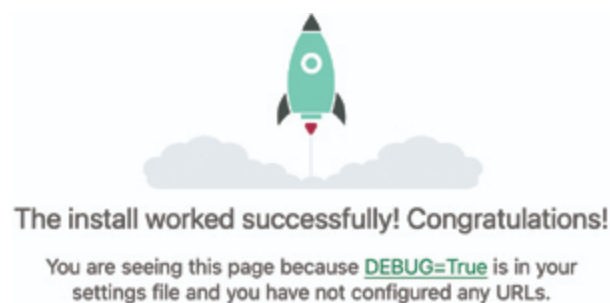
Now, move the project files inside the src directory:

```
rm -rf src/recommendation_system
mv recommendation_system/recommendation_system src/
```

Our project is ready to use. For testing, we will execute the development server:

```
python manage.py runserver
```

If everything is working as expected, you should see the default Django page:



*Figure 1.1: Django default welcome page*

Django REST Framework (DRF) is an open-source framework built on Django. It provides several tools and functionalities to extend Django, allowing developers to build RESTful web services quickly. Let us review some key features of DRF:

- **Browsable API:** The framework comes with a ready-to-use browsable API that allows the users or developers to interact with the API using any API client. This browsable API also acts as a documentation.

- **Serialization:** DRF allows the serialization of ORM models, giving developers a powerful tool to save development time. Serializers are also compatible with lists and dictionaries; all primitive types are supported.
- **Authentication and Permissions:** The framework comes with many different types of authentication methods out of the box, like token-based, session or OAuth. It also offers a flexible permissions system to control access to API resources based on user roles and permissions.
- **Testing:** DRF provides different tools for testing APIs, which integrate with Django's testing framework. It adds support for testing API requests and responses.
- **Throttling:** The framework also supports limiting the rate of requests that our clients are making against our API. Throttling is frequently used to prevent abuse, making our systems more reliable.

Let us proceed to install DRF:

```
rye add.djangorestframework
rye sync
```

Now, we need to add DRF to our installed apps in our projects settings.py file located at `src/recommendation_system/settings.py`. At the end of the list of `INSTALLED_APPS` add `rest_framework`.

## [TDD Concepts and Workflow in Django](#)

In software development, tests are actions and checks performed to ensure that a program does what it is supposed to do and works appropriately.

There are different types of tests. Here is a list of some of them:

- **Unit Tests:** Test individual components or functions in isolation to ensure they work as intended.
- **Integration Tests:** Assess how different parts of the application work together, focusing on the interaction between components.
- **End-to-End Tests:** Simulate real user scenarios to validate the complete system's functionality, data integrity, and interaction with other systems, interfaces, and databases. This type of testing covers the entire application flow, from the initial user input to the final

output, ensuring that the system works as intended in a real-world scenario.

- **Performance Tests:** Measure the application speed under various conditions.

There is a typical pattern to follow when writing tests: the **Arrange-Act-Assert** (AAA) Pattern:

- **Arrange:** Set up the context needed for the test.
- **Act:** Execute the component being tested.
- **Assert:** Verify the outcome is as expected.

Here is an example of a test that follows the AAA pattern:

```
def test_calculate_discount_standard_user():  
    # Arrange  
    user_type = "standard"  
    purchase_amount = 100  
  
    # Act  
    final_amount = calculate_discount(user_type,  
    purchase_amount)  
  
    # Assert  
    expected_amount = 90 # Assuming a 10% discount for standard  
    users  
    assert final_amount == expected_amount, "Standard users  
    should receive a 10% discount."
```

The preceding test first arranges the context by setting the variable **user\_type** and **purchase\_amount** values. Then, it executes the function **calculate\_discount**, which is being tested. Finally, it asserts that the result is the expected amount. If the assertion fails, the string "**Standard users should receive a 10% discount**" will be printed.

Each test should only test one component, which means one function, method, and more. If you need to write a test, keep it focused and simple and make sure the assert section validates the correctness of the behavior being tested, not just a superficial metric like a count. It should confirm the actual state, output, or side effects align with the expected outcomes.

Test Driven Development (TDD) is a software development methodology where the tests are written first. The methodology helps improve the design and quality, forcing the developer to think about using the objects to solve a particular problem. Writing tests first also helps ensure the code covers important scenarios well. Tests also serve as a form of documentation; tests must also be written in a declarative way and be easy to follow.

TDD is based on a simple cycle known as “Red-Green-Refactor”.

1. **RED:** The developer writes tests for required new functionality. The test should fail and indicate that the test correctly detects the absence of this new functionality.
2. **GREEN:** At this step, the developer writes minimal code to make the test pass.
3. **REFACTOR:** When the test passes (green), the developer can clean up the code while ensuring the test still passes. This step usually involves removing duplicate code, improving code readability, or making optimizations. Writing tests in the first step ensures the improvements do not break the functionality.

This simple cycle encourages developers to write tests before writing the code that implements the functionality.

Following TDD requires discipline but brings many benefits, like code quality and maintainability.

## [Building Basic Models in Django](#)

We are ready to write our first Django model, the Movie. It is important on all Django projects to structure the project efficiently since it will allow us to scale and maintain the project in the future. Because of this, we will create a new Django application called Movies. The responsibility of this Django application is to store all information related to movies and allow our users to perform create, read, update and delete (CRUD) operations against this model through a RESTful API. Let us create the movie application:

```
cd src
python manage.py startapp movies
```



Enable the newly created application by adding `movies` to the `INSTALLED_APPS` in the file `src/recommendation_system/settings.py`.

We must create the Movie model, open the file `src/movies/models.py`, and add the following class:

```
from django.db import models

class Movie(models.Model):
    title = models.CharField(max_length=255)
    genres = models.JSONField(default=list)

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

This code snippet defines a `Movie` class representing a Django web framework application model. The model is designed to store information about movies in a database, with fields `title` (title), and a list of `genres` (genres) stored in JSON format. The `__str__` method is overridden to return the movie's title when an instance of the `Movie` class is converted to a string.

**INFO:**

*Django has a powerful migration mechanism for the database. Every time there is a change in the models of your Django project, the framework will detect the changes and create a migration to make the changes in the database. Facilitating collaborative development enables developers to implement changes across various instances and application versions.*

Since we added a new model, we need to make the migrations:

```
$ python manage.py makemigrations
Migrations for 'movies':
  movies/migrations/0001_initial.py
    - Create model Movie
```

Now execute the migrations:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, movies,
  sessions
Running migrations:
```

```
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying admin.0002_logentry_remove_auto_add... OK
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying movies.0001_initial... OK
Applying sessions.0001_initial... OK
```

## Writing Tests Using Pytest

**Pytest** is a popular testing framework for Python. It simplifies how to write tests and has several features that make writing tests easier. In this section, we will install **pytest** and learn how to use it to write some simple tests. We will use **pytest** throughout the book, so knowing how to use it and the framework's features is important.

Beginning with the installation, let us add **pytest** and **pytest-django**:

```
rye install --dev pytest pytest-django
```

In the previous command, we installed **pytest** and **pytest-django** as development dependencies since we don't want to ship testing dependencies in our production environment. We also installed **pytest-django**, which is a **pytest** plugin that provides a set of useful tools for testing Django applications and projects.

We will create some example tests to learn how to write tests using **pytest**. As an example, we are going to test the famous Fibonacci sequence.

Let us first write the tests:

```
import pytest
from your_module import fibonacci # Adjust the import based
on your project structure
def fibonacci(n: int) -> int:
    return 0

@pytest.mark.parametrize("n, expected", [
    (0, 0), # Testing the base case of 0
    (1, 1), # Testing the base case of 1
    (2, 1), # Testing n=2, which is the first case of adding the
    two previous numbers
    (3, 2), # Testing n=3
    (4, 3), # Testing n=4
    (5, 5), # Testing n=5
    (6, 8), # Testing n=6
    (10, 55), # Testing n=10
])
def test_fibonacci(n, expected):
    assert fibonacci(n) == expected
```

The preceding test uses the **pytest** fixture `parametrize`. `Parametrize` allows you to run a single test function multiple times with different sets of arguments with minimal code.

We can now run the tests; you would see that eight failed and one passed. The failed tests are because our implementation of Fibonacci is incomplete. Here is the full implementation:

```
def fibonacci(n: int) -> int:
    """Return the n-th Fibonacci number."""
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    prev, current = 0, 1
    for _ in range(2, n + 1):
        prev, current = current, prev + current
    return current
```

The provided code defines a function `fibonacci(n: int) -> int`, which calculates the **n-th** Fibonacci number. Starting with two initial values, 0 and 1, the function iteratively updates the current and previous values to sum up to the next number in the sequence. This approach efficiently computes the **n-th** number without the overhead of recursion or storing the entire sequence, demonstrating an elegant solution to a problem with deep mathematical roots.

If we now run the tests, we will have all of them green!

**Pytest** has a feature to set up and tear down tests and fixtures. Fixtures allow you to simplify your test code and make it easier to read. Fixtures allows you to set up a database, a test file or a mock and it allows you to use it by injecting the fixtures into your tests.

```
import pytest

@pytest.fixture
def resource() -> Generator[str, None, None]:
    # Set up code before the yield statement
    resource = "Some resource"
    yield resource
    # Tear down code after the yield statement

def test_with_resource_fixture(resource: str) -> None:
    assert resource_fixture == "Some resource"
```

In the preceding example, we are using the decorator `fixture` to indicate to **pytest** that `resource` is a fixture. The fixture can be a generator. When it is a generator, the code above the `yield` will be the setup, and the code after the `yield` will be the tear-down of the fixture. This is usually when the fixture requires the cleanup of some resources. Then, we have our **test\_with\_resource\_fixture**, which uses the `resource` fixture in the parameters. Note that the framework recognizes the parameter name and injects the fixture resource. The test asserts the value of the fixture just to show how fixtures work.

The `conftest.py` file in **pytest** is a special configuration file used by the **pytest** framework to share fixtures, hooks, and other configurations across multiple test files. Unlike regular test files, which contain test functions and are automatically recognized by **pytest**, most of the time, we will have our

fixtures in the `conftest.py` file. Remember that the `conftest.py` could be in multiple directories to distribute your fixtures in the correct modules.

Sometimes, creating data for the tests can become complex and create repetitive code, especially when dealing with model instantiation.

One of the most popular libraries for solving this problem is Factory Boy. Factory Boy is based on Factory Girl, a similar library used in the Ruby on Rails community for generating test data. Factory Boy uses Faker to provide a wide range of data generation options for fields, such as names, addresses, numbers, and more, allowing for the creation of realistic test data.

Let us see an example of how to use Factory Boy to generate new instances of Django model objects.

Install **factory-boy** with **rye** and **sync**:

```
rye add --dev factory-boy
rye sync
```

Create a new file in `src/movies/tests/factories.py` with **MovieFactory** class:

```
from factory import DjangoModelFactory, Faker
from .models import Movie

class MovieFactory(DjangoModelFactory):
    class Meta:
        model = Movie

    title = Faker('sentence', nb_words=4)
    genres = Faker('pylist', nb_elements=3,
        variable_nb_elements=True, value_types=['str'])
```

In this example, **MovieFactory** inherits from **DjangoModelFactory**, linking it to the `Movie` model. The attributes, titles, and genres are defined using Faker providers. When using faker providers, it will generate random data for each field. The title will randomly generate a sentence of four words and genres to a list of variable-length strings. These definitions ensure that each time you create a new `Movie` instance with **MovieFactory**, it will have randomly generated values suitable for testing purposes.

Here are some examples of how to use the factory:

```
unsaved_movie = MovieFactory.build()
```

The **build()** method generates an instance of the model without saving it to the database.

```
saved_movie = MovieFactory.create()
```

The **create()** method generates an instance of the model and saves it to the database, ensuring that the instance is available for database interactions and subsequent queries.

```
movies = MovieFactory.create_batch(5)
```

The **create\_batch()** method generates and saves a specified number of instances to the database.

We will see more examples of usage in [\*Chapter 2, Building and Testing Basic APIs\*](#).

## **Introduction to Git**

A version control system (VCS) is a tool that helps to manage source code changes over time. A VCS records every change of the code with a timestamp and the identity of the developer who made the changes. VCS is a fundamental tool required to collaborate in a team, allowing different lines of development called branches to merge these branches into the current or latest version.

Git, a distributed version control system, facilitates managing small and large projects. Its distributed nature means each developer maintains a full repository copy, synchronizing against external repositories, of which there can be several.

Git's branching capabilities support various workflow strategies, making it an indispensable tool for modern software development.

Let us initialize our project with git:

```
cd recommendation_system
git init
```

With the **git init**, we initialize an empty repository without any changes.

Now let us add the current project files:

```
git add README.md pyproject.toml requirements-dev.lock
requirements.lock src
```

The previous command will add all the specified files and directories to the stage. We cannot commit to create the snapshot:

```
git commit -m 'feat: initialize recommendation system  
codebase'
```

The commit command with the `-m` flag specifies the commit title that indicates the changes done. We can now see the history of changes with the following command:

```
git log
```

The next most common step is to push the changes to an external repository, which is usually shared across all developers:

```
git push
```

Since we don't have a remote repository, the command will fail and you will need to add one like this:

```
git remote add origin  
https://github.com/llazzaro/django_rest_book.git  
git push origin
```

If you want you can clone the code from the GitHub repository:

```
git clone
```

### INFO:

*Conventional Commits is a specification for adding human and machine-readable meaning to commit messages.*

*The message had the following structure:*

*<type>(<scope>): <description>*

- **Type:** Indicates the kind of change the commit makes to the codebase (for example, a feat for new features, fix for bug fixes, docs for documentation changes, and more).
- **Scope (optional):** Provides additional context about what part of the codebase the change affects, making it easier to identify where changes have been made.
- **Description:** A concise explanation of the changes, starting with a capital letter written in the imperative mood.

*Here is a list of common commit types:*

- **feat:** Introduces a new feature to the codebase.
- **fix:** Fixes a bug in your codebase.
- **docs:** Documentation only changes.
- **style:** Changes that do not affect the meaning of the code (white space, formatting, missing semicolons, and more).
- **refactor:** A code change that neither fixes a bug nor adds a feature.

If you want a full list, you can browse the official conventional commits website: <https://www.conventionalcommits.org>.

If you want to create a new branch, use the following command:

```
git switch -c <new-branch-name>
```

You can switch branches with:

```
git switch <branch-name>
```

## Project Introduction

Throughout this book, we will undertake the construction of a recommendation system. A recommendation system is software that tries to predict and suggest items that are likely to interest a user. The results returned by the recommendation system could be movies, books or even social media posts. The main object is to return the most relevant option to the users.

Many methodologies exist to implement a recommendation system:

- **Content-Based:** It compares the item's data and the user profile to recommend relevant results. These systems use item descriptions, genres, authors or any other attributes. The system then uses the items the user has liked or interacted with to make recommendations of similar items.
- **Collaborative:** Uses the past behavior of users in the system rather than the content of the items themselves.
- **Hybrid Systems:** Uses content-based filtering, collaborative filtering and sometimes other approaches to improve recommendation quality.
- **Knowledge-Based:** These systems recommend items based on explicit knowledge about users' needs and preferences. These methodologies



are used when there is not enough data or user preferences.

- **Context-Aware:** Uses additional context, such as the time of day, the user's location, or social context, to provide more appropriate recommendations.

In [\*Chapter 6, Developing a Data Science Project\*](#), we will provide more details on how to build a recommendation system.

Through the chapters of this book, we will build and manage efficient, scalable web APIs for a recommendation system.

## Conclusion

Understanding key concepts is crucial to building efficient, scalable, easy-to-maintain APIs. When building APIs, it is also vital to understand the HTTP standard. Mastering HTTP will make our APIs user-friendly and improve the user experience when integrating your recommendation system with other systems.

Following that, TDD methodology requires strict discipline since it is easier to get coding anxiety. When tests are written first, we will think about how to use our code or API as a user, which will bring quality to our delivered project.

We also got a brief introduction to the project we will build, which we will reiterate in every chapter to incrementally add new features and make it more efficient and scalable.

In the next chapter, we will see how to build and test a basic CRUD API for our movies using TDD development.

## Questions

1. Explain the difference between statically typed and dynamically typed languages with examples.
2. What is the significance of HTTP methods in building RESTful APIs? Name any two HTTP methods and their use cases.
3. Describe the TDD cycle. What does each phase mean?
4. What is the purpose of using f-strings in Python, and how do they differ from traditional string formatting?

5. Explain the concept of 'Duck Typing' in Python with an example.
6. How do Django Migrations facilitate collaborative development in a Django project?
7. What are the principles of REST, and how do they guide the design of a RESTful API?

## **Exercises**

1. **Python Basics:** Create a Python function that reverses a string and uses a **pytest** fixture to test it with multiple test cases.

**Hint:** *Define a fixture that provides a list of strings and their expected reversed forms. Use to apply the fixture.*

2. **Git Basics:** Write a series of Git commands to initialize a new repository, create a new branch called feature/blog, add changes, commit them with a message feat: add blog models and views, and push to a remote repository.

**Hint:** *Start with , follow up with branch creation and switching, then add, commit, and push changes.*

3. **Django Model:** Make a change to the Movie model to add a new attribute. Write a Django migration command to apply the model changes to the database.

**Hint:** *Use or or for the new attributes.*

## **CHAPTER 2**

# **Building and Testing Basic APIs**

### **Introduction**

This chapter will follow the API-first approach and TDD to design an API for our movie model. The API will allow us to create, read, update, and delete our project's movie models. We will follow the red-green-refactor process while building the API. By the end of the chapter, the code will be refactored to provide an optimal solution for implementing our API.

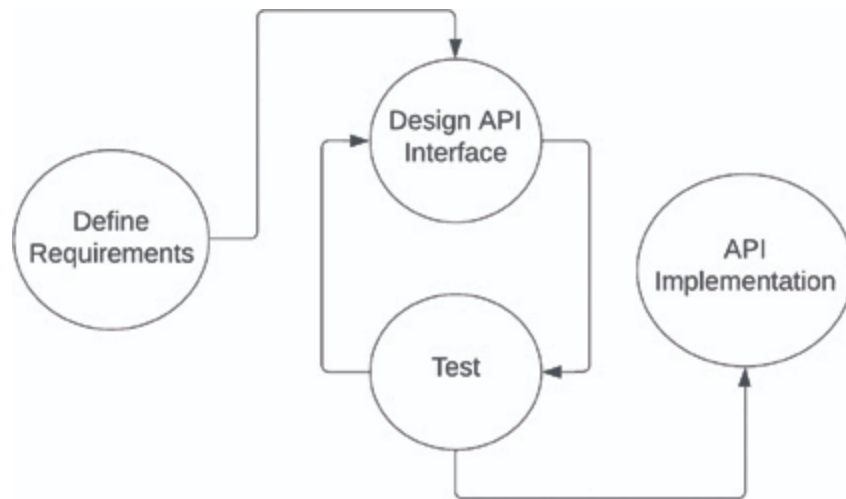
### **Structure**

In this chapter, we will discuss the following topics:

- API-First Design Approach
- Designing Simple APIs
- Testing API Views
- Writing and Testing Serializers
- Developing CRUD Operations
- Refactoring Code with TDD

### **API-First Design Approach**

The API-first approach prioritizes designing and specifying the API before writing any code. This methodology is very effective when you create systems that interact with multiple other systems, services, and clients across different platforms. After iterating on our API design with sufficient stakeholder feedback, we are ready to advance to the implementation stage, ensuring our API aligns closely with user and system requirements. You can get feedback from stakeholders, customers, or users. API designs should include the endpoints, data models, and authentication methods the API will use.



**Figure 2.1:** API-first process

The process starts with understanding the requirements of the API consumers. Once the requirements phase is done, the next step is to design the API. Designing the API involves defining the endpoints, the request/response formats, and the HTTP methods. For this phase, we can use tools or just a document. Next, we can use the API design to create stubs or mocks for testing purposes. We can progress to the implementation stage after sufficiently iterating on our API design. The API implementation stage depends on each team or company but this is the standard development process that ideally should follow TDD as we will do in the next sections.

#### **INFO:**

**Stubs** are simplified, controlled implementations of objects your code under test interacts with. They allow tests to run without relying on the object by returning predefined responses to method calls. Stubs are primarily used to simulate scenarios predictably.

**Mocks** are used to verify interactions between different system parts during a test. They simulate the behavior of natural objects like stubs and keep track of how they are used. This allows you to assert that certain methods have been called with specific arguments, making mocks more about behavior verification than just simulating responses.

## Designing Simple APIs with Django REST Framework

Our first API will be the movies Create, Read, Update, and Delete (CRUD). The movie CRUD API is an excellent starting point for project development, as it is straightforward and aligns with RESTful principles.

Let us start by reviewing our API. Then, we can examine the response and bodies of each API endpoint in more detail.

Operation	HTTP Method	Endpoint	Description
Create	POST	/api/v1/movies	Adds a new movie to the collection.
Read	GET	/api/v1/movies	Retrieves the list of movies.
Read	GET	/api/v1/movies/{id}	Retrieves one movie, using the id to identify the movie.
Update	PUT	/api/v1/movies/{id}	Updates the movie by sending the data in the request's body. The movie to update is identified by its ID.
Delete	DELETE	/api/v1/movies/{id}	Deletes the movie using the id.

*Table 2.1: Movies API*

The API will use token-based authentication using the authorization header. Token-based authentication is a security mechanism for APIs where a client provides a token instead of credentials. The client must include this token in the header of every subsequent request to the server. The server validates the token to authorize or deny the request. Token-based authentication is stateless, meaning the server doesn't need to keep a record of tokens after issuing them. In [Chapter 5, Securing and Scaling Data Science APIs](#), we will add authentication and authorization to our API endpoints.

The movie data model will contain three attributes: the ID, title, and genre.

- **ID:** Integer that uniquely identifies a movie.
- **title:** String with the title of the movie.

- **genres:** List of genres for the movie.

Here is a sample instance of one movie using the JSON format:

```
{
  "id": 2,
  "title": "Midnight Express",
  "genres": ["Crime", "Drama", "Thriller"]
}
```

**INFO:**

*As a simplification, we write an API document to design the API. Ideally, you should use API specifications such as OpenAPI, RAML or similar to design the API. When using API specification, you can use tools to create mocks that simulate your final project.*

In POST operations, the request body mirrors the movie model without the ID, as it's server-generated. Conversely, PUT requests must include the ID in the path to identify the movie to update, rendering the ID in the body redundant. We don't have to give the ID for the POST since the server will provide it. The ID is in the path for the PUT and will be redundant when included in the body.

**INFO:**

*The Movie specification above could be used to get feedback from stakeholders. We, as developers, could think that our API will implement everything required, but once the stakeholders see what they will get, something important needs to be added. What if the movie model should have a year attribute? or any other relevant information for the problem we are solving for them?*

*If stakeholders wait until the code is ready, it could be too late to iterate the API or too expensive.*

For the list endpoint, we will wrap the list of movies in an object that contains metadata about the response, like the total number of movies and the next or previous page link.

Here is the body of our list response:

```
{
```

```
"count": 10,  
"next": "",  
"previous": "",  
"results": [  
  {  
    "id": 2,  
    "title": "Midnight Express",  
    "genres": ["Crime", "Drama", "Thriller"]  
  },  
  ...  
],  
}
```

Using an object instead of a direct list will increase our API's flexibility for future modifications. It will allow us to add more metadata without breaking the API consumers. Also, our API will be more consistent since it will always return an object.

Although directly returning a list could historically expose APIs to vulnerabilities like JSON Hijacking and Cross-Site Script Inclusion (XSSI), contemporary development practices, including rigorous CORS policies and content-type headers, significantly mitigate these risks.

**NOTE:**

*JSON Hijacking is an attack where an attacker can retrieve a JSON response from a web application through a malicious script. An attacker could redefine the Array constructor and include a script from the target domain using a `<script>` tag. Then, the browser executes the script, and the redefined Array constructor captures the JSON array data, allowing the attacker to access sensitive data.*

*Cross-site Script Inclusion attacks are similar to JSON Hijacking and can occur when the API returns sensitive data as a JSON array and doesn't adequately restrict cross-origin requests. An attacker could create a malicious web with a `<script>` tag pointing to a JSON endpoint on the victims' domain. If the user is authenticated on the target domain and the application doesn't correctly validate the request's origin or doesn't use proper content-type headers with charset specifications, the browser might execute the script.*

*These vulnerabilities are much less concerning with modern browsers and secure development practices.*

With our initial API specification, we are ready to continue with our next iteration to add the tests.

## Testing API Views

Following our TDD approach, we will write the tests before the API views. Since we already have a specification, we will follow it to create the tests for each operation.

Let us start with the create operation. In the directory `src/movies/tests`, create the file `test_api.py`.

```
import pytest
from django.urls import reverse
from rest_framework import status

from movies.models import Movie
from .factories import (
    MovieFactory,
)

@pytest.mark.django_db
def test_create_movie(client):
    url = reverse("movies:movie-api")
    data = {"title": "A New Hope", "genres": json.dumps(["Sci-Fi", "Adventure"])}

    response = client.post(url, json=data)

    assert response.status_code == status.HTTP_201_CREATED,
    response.json()
    assert Movie.objects.filter(title="A New Hope").count() == 1
```

The preceding code first imports the required dependencies to execute the tests. **Pytest** is needed to use the decorator `django_db`. The `django_db` is necessary every time the tests need to access the database. This is required to optimize the essential resources when running each test using **pytest**. Then, we import `reverse`, which calculates the API URL. In the following sections, we will add routes with names that can be referenced using the



**reverse** Django function. The status is essential to have a more declarative test code. Finally, we import our **Movie** and the **MovieFactory** model. We need the model to execute queries to verify that the APIs performed the changes in the database.

For the arrange section of the test, we calculate the URL using the **reverse** function and then prepare the data to send to the creation endpoint.

In the act section, we execute the post using the client to submit the movie data as a JSON payload. Note that the post method uses the keyword parameter **json**.

Finally, the assert verifies that the status code. Using the title, it verifies that it exists in the database.

The following test will be retrieved using ID movie:

```
@pytest.mark.django_db
def test_retrieve_movie(client):
    movie = MovieFactory()
    url = reverse("movies:movie-api-detail", kwargs={"pk":
        movie.id})

    response = client.get(url)

    assert response.status_code == status.HTTP_200_OK
    assert response.json() == {
        "id": movie.id,
        "title": movie.title,
        "genres": movie.genres,
    }
```

We use **django\_db** to indicate that the test requires database resources. The arrange section creates a new movie instance. We use the factory to create the model; this will persist the movie in the database and give it an ID value. Then we calculate the movie detail URL using **reverse** and with the keyword arguments **id**, this will translate to **/api/v1/movies/{id}**. The act constitutes the call to the api using the GET method. Finally, we asserted that the status code was 200 OK, and the response contained the expected movie title and genres.

Our next operation to test is the update movie endpoint:

```
@pytest.mark.django_db
```

```
def test_update_movie(client):
    movie = MovieFactory()
    new_title = "Updated Movie Title"
    url = reverse("movies:movie-api-detail", kwargs={"pk":
movie.id})
    data = {"title": new_title}

    response = client.put(url, data=data,
content_type="application/json")
    assert response.status_code == status.HTTP_200_OK,
response.json()
    movie = Movie.objects.filter(id=movie.id).first()
    assert movie
    assert movie.title == new_title
```

In the arrange section, we create a movie using the **MovieFactory**; this will be the movie to update. Next, we set **new\_title** with the title that we want to update. Then, we calculate the API's updated URL using the movie ID and prepare the payload to send (data).

The act uses the put method and sends the payload using the JSON content type.

The assert part verifies that the status code returned is 200 OK. It uses the **Movie.objects.filter(id=movie.id).first()** to load the object from the database and verify that it was correctly updated.

The test for the delete API view:

```
@pytest.mark.django_db
def test_delete_movie(client):
    movie = MovieFactory()
    url = reverse("movies:movie-api-detail", kwargs={"pk":
movie.id})

    response = client.delete(url)

    assert response.status_code == status.HTTP_204_NO_CONTENT
    assert not Movie.objects.filter(id=movie.id).exists()
```

The arrangement will create a new movie using the **MovieFactory**. We will delete this movie using the API in the act section. As always, we prepare the URL.

The act section uses the delete method and the URL to submit the request to the API.

The assert verifies that the 204 status code was returned and that no movie with an ID is in the database.

The last endpoint we need to test is the list endpoint:

```
from django.test import override_settings

@pytest.mark.django_db
@override_settings(REST_FRAMEWORK={'PAGE_SIZE': 10}) #
Override the PAGE_SIZE setting
def test_list_movies_with_pagination(client):
    # Create a batch of movies, adjust the number according to
    # your PAGE_SIZE setting
    movies = MovieFactory.create_batch(10)

    # Define the URL for the list movies endpoint
    url = reverse("movies:api-movie-list")

    # Perform a GET request to the list endpoint
    response = client.get(url)

    # Assert that the response status code is 200 OK
    assert response.status_code == status.HTTP_200_OK

    # Convert the response data to JSON
    data = response.json()

    # Assert the structure of the paginated response
    assert "count" in data
    assert "next" in data
    assert "previous" in data
    assert "results" in data

    # Assert that the count matches the total number of movies
    # created
    assert data["count"] == 10

    # Adjust according to the number of movies created

    # Assert the pagination metadata (if applicable,
    # depending on the number of items and page size)
```

```

# For example, if you expect more items and multiple pages:
# assert data["next"] is not None
# But in this case, if all movies fit on one page:
assert data["next"] is None
assert data["previous"] is None

# Assert that the number of movies in the results
# matches the number of movies created

# This checks the first page of results in case of multiple
pages
assert len(data["results"]) == 10 # Adjust based on your
PAGE_SIZE setting
# Use a set to ensure that the returned movies match the
expected movies
returned_movie_ids = {movie["id"] for movie in
data["results"]}
expected_movie_ids = {movie.id for movie in movies}
assert returned_movie_ids == expected_movie_ids

# Additionally, verify that each movie in the response
contains the expected keys
for movie_data in data["results"]:
    assert set(movie_data.keys()) == {"id", "title", "genres"}

```

The `@override_settings(REST_FRAMEWORK={'PAGE_SIZE': 10})` decorator temporarily sets the **PAGE\_SIZE** configuration in Django's REST framework to 10 for the duration of the test, ensuring that pagination behavior is tested with exactly 10 items per page.

Ten movies are created using **MovieFactory's create\_batch** method in the arrange phase. Then, we calculate the list URL using the reverse Django function.

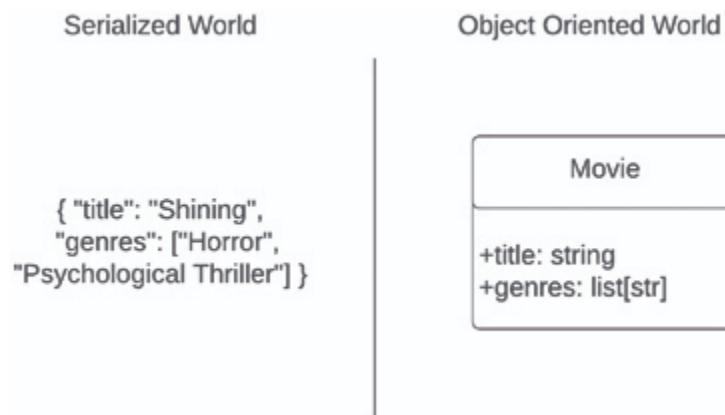
In the act, the test client uses the GET method to retrieve the list of movies.

The assert section verifies the status code and the structure of the list body. Since the body returned by the API is more complex than before, the test asserts the metadata and the results but uses the size of the list of movies and the set to ensure all the expected movies were returned. It also validates that each movie structure contains the expected keys.

## Writing and Testing Serializers

When developing APIs, we can say that there exist two worlds, the object world and the serialized world:

- The object world consists of complex data types, like Django models and query sets. Data is structured optimally for server-side processing, database storage, and business logic writing.
- The serialized world represents data in standardized formats such as JSON or XML that can be easily transmitted over the network and consumed by clients. In this world, data is structured in a standard way between different technologies.



*Figure 2.2: Serialized world versus Object-oriented world*

The role of transforming representation between those two worlds is the serializer, which acts as a bridge. Django REST Framework provides classes to implement this serialization and deserialization. Here is a list of the primary types of serializers that DRF provides:

- **Serializer:** This converts complex data, such as Django models, into Python data types that can be quickly rendered into JSON, XML, or other content types. It is also used for deserialization, turning parsed data into complex types.
- **ModelSerializer:** This is a shortcut for creating serializers that automatically deal with model instances and QuerySets. It uses the model's information to automatically generate a set of fields and includes simple default implementations of the `create()` and `update()` methods.

- **HyperlinkedModelSerializer:** This is similar to **ModelSerializer** but uses hyperlinks to represent relationships rather than primary keys. It provides a way of working with related resources through URL representations, making it more suited for APIs emphasizing discoverability and linked resources.

Let us start with our tests for the serializer. Our first test will verify that we can convert a JSON format to a model instance that is persisted in the database. Create a new test file in the directory **src/movies/tests/test\_serializers.py** with the following content:

```
import pytest
from .models import Movie
from .serializers import MovieSerializer

@pytest.mark.django_db
def test_valid_movie_serializer():
    # Given valid movie data
    valid_data = {
        "title": "Inception",
        "genres": ["Action", "Sci-Fi"]
    }

    # When we create a serializer instance with this data
    serializer = MovieSerializer(data=valid_data)
    # Then, the serializer should be a valid
    assert serializer.is_valid()
    # When saving the serializer
    movie = serializer.save()

    # Then a movie instance should be created with the given
    data
    assert Movie.objects.count() == 1
    created_movie = Movie.objects.get()
    assert created_movie.title == valid_data["title"]
    assert created_movie.genres == valid_data["genres"]
```

In the arrange section of the test, we prepare the dictionary with the movie to serialize. The act section consists of three steps. The first one is to instantiate the serializer using the dictionary. Then, we call the **is\_valid** method provided by DRF. This method will verify that the dictionary

contains valid keys and values. The last part of the act section is to call the method `save`, which will persist the movie into the database.

In the assert section, we verify that a movie exists in the database and contains the data from the dictionary created in the arrange section.

This test verifies that the serializer works as expected with the input and explains how to use the serializer.

Let us write a test to verify that when a required field is missing, the serializer fails when we call the `is_valid` method:

```
@pytest.mark.django_db
def test_invalid_movie_serializer():
    # Given invalid movie data (missing required "title" field)
    invalid_data = {
        "genres": ["Action", "Sci-Fi"]
    }

    # When we create a serializer instance with this data
    serializer = MovieSerializer(data=invalid_data)

    # Then serializer should not be valid
    assert not serializer.is_valid()

    # It should contain an error message for the missing "title"
    # field
    assert "title" in serializer.errors
```

In the arrangement, the `invalid_data` dictionary contains only the genres, but the title is missing. The act section specifies the `MovieSerializer` with the dictionary, creating a serializer.

In the assertion, we verify that the `is_valid` method returns `False` and that the title key exists in the errors, which will explain why the serialization failed.

Finally, we must test the reverse, converting the model to a dictionary.

```
@pytest.mark.django_db
def test_serialize_movie_instance():
    # Given a movie instance
    movie = Movie.objects.create(title="Inception", genres=
        ["Action", "Sci-Fi"])
```

```

# When we serialize the movie
serializer = MovieSerializer(movie)

# Then the resulting JSON data should contain the movie's
details
assert serializer.data == {
    "id": movie.id,
    "title": movie.title,
    "genres": movie.genres
}

```

In the arrangement, we use the Django objects manager to create a new movie with the title Inception and the genres Action and Sci-Fi.

In the act, the **MovieSerializer** is instantiated with the movie object.

In the assert, we verify that the data attribute contains the dictionary with the expected movie ID, title and genres.

You typically inherit from the DRF Serializer or **ModelSerializer** class to write a serializer. Let us see an example for our Movie model:

```

class Movie(models.Model):
    title = models.CharField(max_length=255)
    genres = models.JSONField(default=list)

    def __str__(self):
        return self.title

```

Create a new file in the new directory **src/movies/serializers.py** with the **MovieSerializer**:

```

from rest_framework import serializers
from .models import Movie

class MovieSerializer(serializers.Serializer):
    id = serializers.IntegerField(label="Movie ID",
    required=False)
    title = serializers.CharField(max_length=255)
    genres = serializers.ListField(
        child=serializers.CharField(max_length=100),
        allow_empty=True,
        default=list
    )

```



```

def create(self, validated_data):
    """
    Create and return a new `Movie` instance, given the
    validated data.
    """
    return Movie.objects.create(**validated_data)

def update(self, instance, validated_data):
    """
    Given the validated data, update and return an existing
    `Movie` instance.
    """
    instance.title = validated_data.get("title",
    instance.title)
    instance.genres = validated_data.get("genres",
    instance.genres)
    instance.save()
    return instance

```

In the preceding code, we inherit from the DRF Serializer class and specify its attributes and type. Using serializer types, the framework can transform the data using the appropriate implementation. We also implemented the creation and update method to persist and update the object in the database as required by the tests. If we execute the tests, we will get a green pass test. By the end of this chapter, we will refactor our solution to simplify our code using the **ModelSerializer**.

Let us execute the tests:

```
rye run pytest
```

### INFO:

*Django rest framework serializers have field types that implement how to serialize and deserialize the fields. DRF ships with many field types for the most common data structures:*

- **CharField:** For string fields.
- **IntegerField:** For integer fields.
- **BooleanField:** For true/false fields.

- **DateField:** For date fields.
- **EmailField:** For email fields.
- **URLField:** For URL fields.

*Serializer has more options that can be used:*

- **fields:** Specifies the fields that should be included in the serialized output. It can be a list of field names you want to include in your serialization.
- **exclude:** Indicates the fields that should be excluded from the serialized output. It is the opposite of fields, allowing you to specify which fields to ignore.
- **read\_only\_fields:** Specifies the tuple of field names that should be treated as read-only.

## Developing CRUD Operations

In this section, we will implement all the endpoints we defined in our API contract at the beginning of this chapter. We already have the tests for these endpoints that are failing. We need to make our tests pass.

Let us create a new file in `src/movies/api.py` with the API implementation.

```
from rest_framework import status, views
from rest_framework.response import Response
from .models import Movie
from .serializers import MovieSerializer

class MovieAPIView(view.APIView):
    def post(self, request):
        serializer = MovieSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
            status=status.HTTP_400_BAD_REQUEST)
```

Our API utilizes class-based views inherited from the REST framework's `APIView`. The `APIView` allows for handling different HTTP methods, such as `get`, `post`, `put`, `patch`, and `delete`. Each method corresponds to the HTTP method. The `MovieAPIView` implements the `post` method and instantiates the serializer with the requested data. If the serializer is valid, we will save the movie and return the appropriate status code to indicate that the resource was created. When the data is invalid, we return a `Bad request` with the error to let the API consumer know why the request failed.

**INFO:**

*Django class-based views offer a modular and reusable approach to handling web requests. By encapsulating view logic within a class, developers can leverage the power of object-oriented programming to create well-organized and scalable web applications. These views simplify the code for common patterns, such as displaying forms or lists of objects, by providing generic classes that can be extended and customized through inheritance.*

Next, we will add the read API views, the `get by id` and the `list`:

```
class MovieListView(views.APIView):
    def post(self, request):
        ...
    def get(self, request, pk=None):
        if pk:
            # Retrieve a single movie
            movie = get_object_or_404(Movie, pk=pk)
            serializer = MovieSerializer(movie)
            return Response(serializer.data)
        else:
            # List all movies
            movies = Movie.objects.all()
            serializer = MovieSerializer(movies, many=True)
            return Response({
                "count": len(serializer.data),
                "results": serializer.data,
                "next": None,
                "previous": None,
```

```
    })
```

The get method implements both the list and the get by ID. If the pk is not None, we check if the movie with the pk exists in the database or return 404. When the movie exists, we instantiate the serializer with the object from the database, which will be returned as a response.

When the pk is not provided, the code retrieves all the objects from the database, and the serializer is instantiated with the parameter **many=True**. When using many as true, the serializer will serialize the QuerySet to a list of dictionaries. We also include metadata, count, next, and previous links for the list response.

To follow best practices, we would split the two branches in the get method into two separate methods. This is left as an exercise to the reader.

Then, we will extend our class with the implementation of the put method that will be used for updating the movie by pk:

```
def put(self, request, pk):
    movie = get_object_or_404(Movie, pk=pk)
    serializer = MovieSerializer(movie, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(status=status.HTTP_200_OK)
    return Response(serializer.errors,
                    status=status.HTTP_400_BAD_REQUEST)
```

Initially, the method retrieves the movie by employing **get\_object\_or\_404**, returning the object if present or raising a 404 error otherwise. When the object exists, the serializer is instantiated with the request data. We check if the serializer contains valid data. If the serializer validates the data payload, we save the object to the database and return a 200. If the serializer fails to validate the payload in the request, the API returns a 400 bad request to inform the API consumer that it cannot operate.

Finally, we add the delete method:

```
def delete(self, request, pk):
    movie = get_object_or_404(Movie, pk=pk)
    movie.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)
```

The delete method retrieves the object as we did in the other methods, and then we use the movie delete method to delete the movie from the database. The API returns the status 204 with no content.

We must still define the **src/movies/urls.py** to add the routing to our API view:

```
from django.urls import path
from movies.api import MovieAPIView
app_name = "movies" # Define the application namespace
urlpatterns = [
    path("movies/", MovieAPIView.as_view(), name="movie-api"),
]
```

We need to add the URLs to the Django project, edit the file **src/recommendation\_system/urls.py**, and include the movies app API URL:

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/", include("movies.urls")),
]
```

Let's check that all of our tests pass:

```
rye run pytest -vvv -s
>> collected 8 items
>> src/movies/tests/test_api.py....
>> src/movies/tests/test_serializers.py ...
>> ===== 8 passed in 0.18s
=====
```

## [Refactoring Code with TDD](#)

We have been following TDD and implemented our first iteration of the movie API. To this point, our tests are green. However, our code could be more optimal and implement all the features.

Let us start with the **MovieSerializer**. Our current implementation inherits from DRF Serializer but it can be simplified by just using **ModelSerializer**:

```
from rest_framework import serializers
from .models import Movie

class MovieSerializer(serializers.ModelSerializer):
    class Meta:
        model = Movie
        fields = ["id", "title", "genres"]
```

Now, we need to re-run the tests to make sure nothing was broken:

```
rye run pytest
>> collected 8 items
>> src/movies/tests/test_api.py....
>> src/movies/tests/test_serializers.py ...
>> ===== 8 passed in 0.18s
=====
```

Success! Our test passed, and our code is more straightforward. Let us continue with our API implementation, which is far from simple. The list endpoint lacks pagination.

Django Rest Framework (DRF) offers powerful generic views that abstract standard API functionality into reusable classes. These views significantly reduce the code needed to create APIs.

- **ListAPIView**: A read-only endpoint for listing a collection of model instances.
- **CreateAPIView**: Provides a post-method handler to create a new model instance from the request data.
- **RetrieveAPIView**: A read-only endpoint that retrieves a specific model instance by providing the primary key.
- **DestroyAPIView**: Allows for the deletion of a specific model instance identified by the primary key
- **UpdateAPIView**: This view handles partial or complete updates to a model instance. It supports the PUT (for complete updates) and PATCH (for partial updates) methods.

The DRF framework also provides classes that combine operations. For example, the **ListCreateAPIView** combines the creation and the list features. As another example, the view **RetrieveUpdateDestroyAPIView** will combine views that can retrieve, update, or delete a model instance.

When using generic views with DRF, you usually need to define the queryset and the **serializer\_class** **class attributes**. The queryset should be used to retrieve objects from the database, and the serializer class should be used to serialize and deserialize input and output data for the model associated with the queryset.

Let us refactor our API views to use the generics and simplify our API code:

```
from rest_framework import generics
from .models import Movie
from .serializers import MovieSerializer

# For listing all movies and creating a new movie
class MovieListCreateAPIView(generics.ListCreateAPIView):
    queryset = Movie.objects.all().order_by('id')
    serializer_class = MovieSerializer

# For retrieving, updating, and deleting a single movie
class
MovieDetailAPIView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Movie.objects.all()
    serializer_class = MovieSerializer
```

We also need to change our `urls.py` of the movies application, open the file **src/movies/urls.py** and update the URL patterns:

```
from django.urls import path
from .views import MovieListCreateAPIView, MovieDetailAPIView

urlpatterns = [
    path('movies/', MovieListCreateAPIView.as_view(),
        name='movie-list'),
    path('movies/<int:pk>/', MovieDetailAPIView.as_view(),
        name='movie-detail'),
]
```

Executing the tests reveals most pass with a ‘green’ status, except for the list API test, which fails due to missing metadata.

The `ListCreateAPIView` has pagination, but we need to configure it to enable them. Open the project settings file `src/recommendation_system/settings.py` and add the DRF pagination settings:

```
...
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.PageNumber Pagination',
    'PAGE_SIZE': 10,
}
```

After enabling pagination, all tests should pass:

```
rye run pytest
>> collected 8 items
>> src/movies/tests/test_api.py....
>> src/movies/tests/test_serializers.py ...
>> ===== 8 passed in 0.18s
=====
```

After the API refactor, we have a simple code and pagination in our API thanks to the DRF framework generics API views.

In [\*Chapter 5, Securing and Scaling Data Science APIs\*](#), we will add the token-based authentication to our API.

## Conclusion

In this chapter, we started with an API design to understand what we have to build: a CRUD API for our movie model.

We started by writing the tests for our API, which were based on our API design. Then, we introduced the concepts of serialization and deserialization, which are very important to understand when building APIs. Once we have our tests developed, we continue with our first version of the serializers.

We then continued developing the API endpoints and made our tests green, as we did with the serializers.

By the end of this chapter, we refined our initial API iteration into a more efficient implementation, leveraging the framework's robust features.



In the next chapter, we will extend our database model and add APIs that will be the foundations of the recommendation system.

## Questions

1. What is the primary goal of adopting an API-first design approach in software development?
2. How does the red-green-refactor process work in Test-Driven Development (TDD)?
3. Describe how feedback influences the API design process in an API-first approach.
4. Discuss the importance of model serializers in Django REST Framework.
5. What is pagination, and why is it necessary in APIs dealing with large datasets?
6. Why is it essential to include authentication methods in API designs?

## Exercises

1. Use cURL to verify that the POST endpoint correctly adds a new movie to the database and returns a 201 status code.
2. Extend the movie model by including a new year attribute. Update the API endpoints to support this new attribute.
3. Implement CRUD API Endpoints for a book model:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=255)
    author = models.CharField(max_length=100)
    isbn = models.CharField(max_length=13, unique=True)
    publication_date = models.DateField()
```

## **CHAPTER 3**

# **Data Models and Processing in Data Science**

## **Introduction**

This chapter explores data models and processing in data science, focusing on creating a personalized movie recommendation system. We will extend a basic movie model to include user profiles and watch history. Using SPARQL, we will extract movie data from Wikidata. Finally, we will create a generic API for data ingestion that supports multiple file formats.

## **Structure**

In this chapter, we will discuss the following topics:

- Movie Recommendation System
- Designing Data Models for Data Science Projects
- Extending the Models
- API Architecture
- Creating Endpoints for the User Profile
- Understanding SPARQL
- Getting the Data for Our Project
- Creating and Testing APIs for Data Ingestion

## **Movie Recommendation System**

In a digital era where streaming platforms and digital libraries offer access to thousands of movies, users often need help with the number of choices available. This “paradox of choice” makes it increasingly challenging for users to discover movies that align with their preferences. We need a way to provide personalized movie recommendations to users.

For this purpose, we will extend the movies application, defined in [Chapter 1, Django REST and TDD Essentials](#), to include a user profile. The user profile will be used to gather information about user preferences and a watch history. With the models we are going to add in this chapter, we will build the foundations of our recommendation system.

## **Designing Data Models for Data Science Projects**

Designing data models involves structuring your data to ensure organization and accessibility, which allows effective processing, analysis, and the generation of insights. Understanding the problem you are trying to solve is crucial for designing the appropriate model. Identifying the data sources is another critical step that must be completed before developing our data model.

### **INFO:**

*A data model defines the structure of data elements and their relationships. It is crucial for organizing data in a way that is both efficient and meaningful for specific applications, like a movie recommendation system.*

Data sources include databases, external APIs, sensors, user inputs, and more. Knowing our data sources will help us effectively develop better integration and data.

Wikidata (<https://www.wikidata.org/>) is a free, collaborative, multilingual knowledge base that collects structured data. Launched by the Wikimedia Foundation in 2012, it is a central storage for structured data. The platform allows users and machines to access, manage, and share data.

Wikidata enables data download in various formats, such as JSON, RDF, and XML. It also allows us to connect to an API or even use SPARQL. Subsequent sections will explain how SPARQL facilitates the extraction of data for our project.

### **INFO:**

**SPARQL (SPARQL Protocol and RDF Query Language):** *SPARQL is a query language capable of retrieving and manipulating data stored in Resource Description Framework (RDF) format. It is used to query databases that store information as RDF triples.*

**RDF (Resource Description Framework):** It is a standard model for data interchange on the web. It allows data to be linked to and/or integrated with other data with minimal effort. RDF data is represented in triples, comprising a subject, predicate, and object.

**Wikidata:** Wikidata is a free, collaborative, multilingual database that provides structured data to support Wikipedia, other Wikimedia projects, and beyond. It is a source of open data that can be used for various applications, including data science projects.

## Extending the Models

We can extend the movie model defined in [Chapter 1, Django REST and TDD Essentials](#), with all this information. Let us recap the model:

```
from django.db import models

class Movie(models.Model):
    title = models.CharField(max_length=255)
    genres = models.JSONField(default=list)

    def __str__(self) -> str:
        return self.title
```

We will extend the model with new attributes to store the additional data. The following are the new columns we are going to add:

- **country:** It contains the ISO 3166 Alpha-2 country code.
- **extra\_data:** A dictionary that stores extra information, including directors, actors, or similar information about the movie.
- **release\_year:** Date of the release. We only record one, but sometimes it depends on the country. We will only allow years from 1888 to the current year.

```
from django.db import models
from django.core.validators import MinValueValidator,
MaxValueValidator
import datetime
class Movie(models.Model):
    title = models.CharField(max_length=255)
    genres = models.JSONField(default=list)
```

```

country = models.CharField(max_length=100, blank=True,
null=True)
extra_data = models.JSONField(default=dict)
release_year = models.IntegerField(
    validators=[
        MinValueValidator(1888),
        MaxValueValidator(datetime.datetime.now().year)],
    blank=True,
    null=True
)

def __str__(self) -> str:
    return self.title

```

The model's **Meta** class, specifying **unique\_together** ensures that each combination of **title**, **country**, and **release\_year** is distinct within the database. This means that there cannot be two **Movie** entries with the same title, country, and **release\_year**.

Since we changed our model, we need to generate the migration for it:

```

rye run python src/manage.py makemigrations
rye run python src/manage.py migrate

>> Running migrations:
>> ...
>> Applying
movies.0002_movie_country_movie_extra_data_movie_release_year...
OK

```

The next step is to create a model to store the user's movie preferences. Open the **src/movies/models.py** and add the new **UserMoviePreferences** model:

```

from django.conf import settings
from django.db import models
from django.db.models import JSONField

class UserMoviePreferences(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
on_delete=models.CASCADE, related_name="movie_preferences")
    preferences = JSONField(default=dict, help_text="Stores user
preferences for movies like genres, directors, etc.")

```

```
watch_history = JSONField(default=list, help_text="Stores  
information about movies the user has watched.")
```

```
def __str__(self) -> str:  
    return f"{self.user.username}'s Movie Preferences"
```

The **UserMoviePreferences** has a one-to-one relationship with the user model. We use **settings.AUTH\_USER\_MODEL** since Django allows us to customize the user model, which will ensure that it refers to the correct user model.

The preferences model has two JSON fields, one for **preferences** and another for **watch\_history**. The field with preferences will store genres, directors, countries or potential future fields that we can add in upgrades to our recommendation system.

**INFO:**

*Since this model will only store preferences related to movies and our project scope will be a recommendation system, we will add our new model to the movie application. In other scenarios, you can reconsider creating a new Django application for users and adding the profile model to this application.*

Choosing JSON fields for their flexibility, we enable the system to accommodate the introduction of new fields in the future. Otherwise, we need to redesign the database schema whenever you want to add a new preference. Using JSON will make potential integration with a frontend easier since the query will not need to do expensive joins. Many machine learning and analytics tools can ingest JSON data directly. This compatibility can make analyzing user behavior, training recommendation algorithms, or integrating third-party analytics services smoother.

We changed our model, generated the migration for it and migrated the database:

```
rye run python src/manage.py makemigrations  
rye run python src/manage.py migrate  
>> Running migrations:  
>> Applying movies.0003_usermoviepreferences... OK
```

**INFO:**

*The Django framework supports different types of relationships between models. Relationships mirror real-world associations between models, allowing databases to store interrelated data in a structured manner.*

**One-to-one** relationships could connect each model to a unique model using Django's `OneToOneField`. This is useful when the relationship's cardinality is a maximum of 1 for each side.

**Many-to-one** relationships link multiple models to one model, facilitated by Django's `ForeignKey`.

**Many-to-many** relationships are realized through Django's `ManyToManyField`, which associates models with multiple models and vice versa.

*This book prioritizes using JSON fields for data science due to their flexibility with dynamic and complex data over traditional relational database relationships. JSON fields support rapid iteration and diverse data needs, fitting data science's evolving requirements. We advocate for a balanced approach, combining JSON's adaptability with relational models' integrity, to effectively meet varied data management demands.*

*It is important to know that when a JSON object is large, it may exceed the default page size. This can lead to issues because PostgreSQL has to store the excess data in TOAST (The Oversized-Attribute Storage Technique) tables. This is also something to consider when using JSON fields.*

Create a new file to add a Django signal to the movies application. Let us call it `src/movies/signals.py` and add the new `post_save` signal for the User model:

```
from typing import Type
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.contrib.auth import get_user_model
from django.db.models.base import ModelBase
from movies.models import UserMoviePreferences

User = get_user_model()

@receiver(post_save, sender=User)
def create_or_update_user_movie_preferences(sender:
Type[ModelBase], instance: User, created: bool, **kwargs) ->
```

None:

```
if created:
    UserMoviePreferences.objects.create(user=instance)
else:
    instance.movie_preferences.save()
```

The preceding code sets a new Django signal that will be called after the User model is saved to the database. We use `get_user_model` since the Django framework allows us to configure a custom User model. The function `create_or_update_user_movie_preferences` will check if the User was created. When the parameter `created` is True, it will create a new instance empty of the `UserMoviePreferences` associated with the user just created. Otherwise, it will save the user's `movie_preferences` to avoid losing data.

#### INFO:

*Django signals serve as mechanisms for notifying decoupled applications about specific actions occurring within the scope of a Django project. Signals allow specific senders to inform a set of receivers when particular events happen.*

*While Django signals offer a tempting route for handling various background tasks, logging, or triggering actions upon certain events, it is essential to use them judiciously. Overuse or misuse of signals can lead to harder debugging and maintaining code. Signals can obscure the flow of a program, making it difficult to trace through the logic and pinpoint the origins of bugs.*

*However, it is important to note that signals are not triggered in all cases; for instance, they won't fire if a row is added directly to the database or when using Django bulk operations like **bulk\_create**.*

## API Architecture

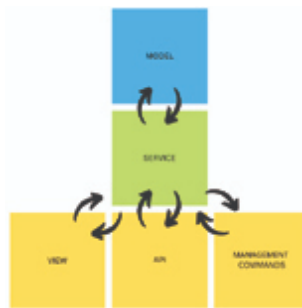
We need to architect the communication to our models before diving deep into getting the data for our project. Django employs the **Model-View-Template** (MVT) architecture pattern, efficiently structuring web applications to separate data handling, business logic, and presentation layers. In MVT, the **Model** represents the data structure, directly managing



the application's data, logic, and rules. The **View** acts upon the model and the template, processing incoming requests, making calls to the model for data, and deciding which template to render. The **Template** is the presentation layer responsible for formatting and displaying the data to the user. This separation of concerns allows developers to modify the presentation of data independently of the business logic behind it, facilitating a straightforward and efficient development process.

There are usually two approaches when using Django: use the models directly or add a service layer. The **service layer** works like an interface and can be helpful for future project maintenance or migration to another technology. The service layer is not about changing the database. The service layer is about where to write the business logic and how other Django applications will interact with the recommendation system. If you have other Django applications that will use the recommendation system not through the API, you should use an interface instead of database models.

The Service Layer, an additional component not explicitly defined in MVT, acts as a middleman between the Views and the Models.



*Figure 3.1: MVT patterns with a service layer*

This layer contains business logic that doesn't naturally fit within Models or Views, promoting reusability and separation of concerns. Abstracting complex operations into services makes the application more modular, easier to maintain, and better structured for scaling.

## [Creating Endpoints for the User Profile](#)

Let us add new endpoints to our API to add preferences and movies to the watch list. Our first step will be to design the API and then add tests.

The new APIs that we will add are four endpoints:

Operation	HTTP Method	Endpoint	Description
Create Preferences	POST	/api/v1/movies/users/{user_id}/preferences/	Adds new movie preferences for the user.
Read Preferences	GET	/api/v1/movies/users/{user_id}/preferences/	Retrieves the user's movie preferences.
Create Watch History	POST	/api/v1/movies/users/{user_id}/watch-history/	Adds a new movie to the user's watch history.
Read Watch History	GET	/api/v1/movies/users/{user_id}/watch-history/	Retrieves the user's watch history.

**Table 3.1:** User preferences and watch list

The API employs token-based authentication through the authorization header. In [Chapter 5, Securing and Scaling Data Science APIs](#), we will add authentication and authorization to our API endpoints.

**INFO:**

*In the context of building APIs, particularly those involved in data collection and user privacy, ethical considerations extend beyond conventional technical challenges.*

*While APIs serve as the backbone for data exchange and system integration, their design and implementation must be rooted in ethical practices to ensure user trust and compliance with global privacy standards.*

While the discussion on ethical considerations in API development is crucial, it falls outside the primary focus of this book; however, readers are encouraged to seek additional resources on this topic to ensure comprehensive understanding and compliance.

The payload for adding new preferences using the POST method will be:

```
{
  "new_preferences": {
```

```
    "genre": "comedy",
    "director": "Quentin Tarantino",
    "actor": "Samuel L. Jackson",
    "year": "1994"
  }
}
```

The response body of the GET operation will contain all the preferences added before:

```
{
  "genre": ["comedy"],
  "director": ["Quentin Tarantino"],
  "actor": ["Samuel L. Jackson"],
  "year": ["1994"]
}
```

The creation endpoint is restricted to accept only genre, director, actor, and year. The API will reject any other preference.

As for the watch history, here is an example payload for adding a movie to the watch list using the POST method:

```
{
  "id": 1
}
```

The payload uses the movie ID to add a new movie to the user's watch list. Here is how to send the payload using curl:

```
curl -X POST http://localhost:8000/api/v1/movies/user/1/watch-
history \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 1
  }'
```

The GET endpoint for getting the watchlist will return the following structure:

```
curl -X GET
http://localhost:8000/api/v1/movies/users/123/watch-history
```

The response of the API will contain the watch history in the following JSON:

```
{
  "status": "success",
  "watch_history": [
    {
      "title": "Midnight Express",
      "year": "1978",
      "director": "Alan Parker",
      "genre": "Drama, Crime",
      "created_at": "2024-03-21T12:45:00Z"
    },
    {
      "title": "The Shining",
      "year": "1980",
      "director": "Stanley Kubrick",
      "genre": "Horror, Drama",
      "created_at": "2024-03-22T15:30:00Z"
    }
  ]
}
```

Having designed our new endpoints, we are ready to write the tests to verify that our API implementation meets the API design above. Create a new file with the tests for the **api** in **src/movies/tests/test\_api.py** and add the following tests:

```
import pytest
from rest_framework.test import APIClient
from django.urls import reverse
from .factories import UserFactory
@pytest.mark.django_db
@pytest.mark.parametrize(
    "new_preferences, expected_genre",
    [
        ({"genre": "sci-fi"}, "sci-fi"),
        ({"genre": "drama"}, "drama"),
        ({"genre": "action"}, "action"),
        ({"genre": "sci-fi", "actor": "Sigourney Weaver", "year":
"1979"}, "sci-fi"),
    ]
)
```

```

)
def test_add_and_retrieve_preferences_success(new_preferences,
expected_genre):
    user = UserFactory()
    client = APIClient()
    preferences_url = reverse("user-preferences", kwargs=
{"user_id": user.id})

    # Add new preferences
    response = client.post(preferences_url, {"new_preferences":
new_preferences}, format="json")
    assert response.status_code in [200, 201]

    # Retrieve preferences to verify
    response = client.get(preferences_url)
    assert response.status_code == 200
    assert response.data["genre"] == [expected_genre]

```

The test checks the API by adding a preference and then fetching it, in this case, the genre. It uses parameterization to assess multiple preference scenarios, ensuring versatility in data handling. A user is simulated using Factory Boy, and Django Rest Framework's **APIClient** performs POST and GET requests to mimic real user interactions. The test validates both the process of storing preferences and the integrity of data retrieval.

Testing should also cover scenarios with invalid payloads, such as empty or incorrect fields.

```

import pytest
from rest_framework.test import APIClient
from django.urls import reverse
from .factories import UserFactory

@pytest.mark.django_db
@pytest.mark.parametrize(
    "new_preferences",
    [
        ({}), # Empty preferences
        ({"genreee": "comedy"}), # Invalid field
    ]
)

```

```
def test_add_preferences_failure(new_preferences):
    user = UserFactory()
    client = APIClient()
    preferences_url = reverse("user-preferences", kwargs=
{"user_id": user.id})

    # Attempt to add new preferences
    response = client.post(preferences_url, {"new_preferences":
new_preferences}, format="json")
    assert response.status_code == 400, response.json()
```

This test verifies that the API appropriately responds with a 400 status code when handling invalid user preference inputs. In the arrange sections, a mock user is created with Factory Boy's **UserFactory**, and requests are made using DRF's **APIClient** to the user preferences endpoint. By checking for a 400 response to these malformed requests, the test verifies the API's robust input validation:

```
import pytest
from django.urls import reverse
from rest_framework.test import APIClient
from .factories import UserFactory, MovieFactory

@pytest.mark.django_db
def test_add_and_retrieve_watch_history_with_movie_id() ->
None:
    user = UserFactory()
    client = APIClient()
    watch_history_url = reverse("user-watch-history", kwargs=
{"user_id": user.id})

    # Create movie instances using the MovieFactory
    movie1 = MovieFactory(title="The Godfather",
release_year=1972, directors=["Francis Ford Coppola"], genres=
["Crime", "Drama"])
    movie2 = MovieFactory(title="Taxi Driver", release_year=1976,
directors=["Martin Scorsese"], genres=["Crime", "Drama"])

    # Add movies to watch history using their IDs
    for movie in [movie1, movie2]:
```

```

    response = client.post(watch_history_url, {"id": movie.id},
                           format="json")
    assert response.status_code == 201

    # Retrieve watch history to verify the addition
    response = client.get(watch_history_url)
    assert response.status_code == 200
    # This assumes your response includes the movie IDs in the
    watch history
    retrieved_movie_ids = [item["title"] for item in
                           response.data["watch_history"]]
    for movie_title in [movie1.title, movie2.title]:
        assert movie_title in retrieved_movie_ids

```

The tests arrange section starts with creating user and movie instances with Factory Boy, ensuring the presence of predefined movie records in the database. During the action phase, we perform POST requests to add each movie to the user's watch history by referencing the movie IDs. In the assertion phase, we retrieve the watch history for the user to verify the successful addition of the movies. We assert that the response status code indicates success and that the movie IDs of the added movies match those retrieved, confirming the API's expected behavior and the integrity of the watch history feature.

```

import pytest
from django.urls import reverse
from rest_framework.test import APIClient
from .factories import UserFactory

@pytest.mark.django_db
def test_add_invalid_movie_id_to_watch_history() -> None:
    # Arrange: Create a user instance using Factory Boy
    user = UserFactory()
    client = APIClient()
    watch_history_url = reverse("user-watch-history", kwargs=
                                {"user_id": user.id})

    # Act: Attempt to add a non-existent movie to the user's watch
    history
    invalid_movie_id = 99999 # Assuming this ID does not exist in
    the database

```

```

response = client.post(watch_history_url, {"movie_id":
invalid_movie_id}, format="json")

# Assert: Check for a 400 Bad Request response
assert response.status_code == 400, "Expected a 400 Bad
Request response for an invalid movie ID"

```

In the arrange phase, we set up the necessary environment by creating a test user with Factory Boy and defining the endpoint URL using Django's **reverse** function. The act section attempts to add a movie to the user's watch history using an ID that does not correspond to any film in the database. We validate the API's response to this action in the assert phase, expecting a **400 Bad Request** status. This assertion confirms that the API appropriately handles invalid data by rejecting requests and providing feedback through an HTTP status code.

The next step is to define the serializers and the API endpoints. Open the **src/movies/serializers.py** and the following serializers:

```

class PreferencesDetailSerializer(serializers.Serializer):
    genre = serializers.CharField(max_length=100,
allow_blank=True, required=False)
    director = serializers.CharField(max_length=100,
allow_blank=True, required=False)
    actor = serializers.CharField(max_length=100,
allow_blank=True, required=False)
    year = serializers.IntegerField(min_value=1900,
max_value=2099, required=False, allow_null=False)

    def validate(self, data: dict[str, Any]) -> dict[str, Any]:
        # Check if all fields are empty or not provided
        if all(value in [None, ""] for value in data.values()):
            raise serializers.ValidationError(
                "At least one preference must be provided."
            )
        return data

class AddPreferenceSerializer(serializers.Serializer):
    new_preferences = PreferencesDetailSerializer()

```

**PreferencesDetailSerializer** validates individual preferences like genre, director, actor, and year, ensuring they meet specific criteria, such as length



or value range, and allowing them to be optional. The method **validates** and ensures that the payload is not an empty dictionary and at least one field exists.

**AddPreferenceSerializer** embeds the **PreferencesDetailSerializer** serializer within the **new\_preferences** field, enabling structured and validated nesting of preference data. This setup allows for precise and flexible handling of user preferences and ensures data integrity through detailed validation rules.

Next, we will add the serializer for adding a movie to the watch list:

```
from rest_framework import serializers
from django.core.exceptions import ObjectDoesNotExist
from .models import Movie

class AddToWatchHistorySerializer(serializers.Serializer):
    id = serializers.IntegerField()

    def validate_id(self, value: int) -> int:
        """
        Check if the id corresponds to an existing movie.
        """
        if not Movie.objects.filter(id=value).exists():
            raise serializers.ValidationError("Invalid movie ID. No
            such movie exists.")
        return value
```

The serializer **AddToWatchHistorySerializer** will validate incoming requests to add movies to a user's watch history. The serializer checks that each submitted ID corresponds to an existing movie in the database. The **validate\_id** method checks the movie's existence. If the movie isn't found, a **ValidationError** is raised.

Finally, we need two more serializers, one for the response for the endpoint to obtain the preferences and another for the watch list.

**INFO:**

*We should write the tests before writing the serializers, but we will let it be an exercise for the reader.*

```
class PreferencesSerializer(serializers.Serializer):
```

```

genre = serializers.ListField(child=serializers.CharField(),
required=False)
director =
serializers.ListField(child=serializers.CharField(),
required=False)
actor = serializers.ListField(child=serializers.CharField(),
required=False)
year = serializers.ListField(child=serializers.CharField(),
required=False)

```

The **PreferencesSerializer** class is a serializer designed to handle user preferences. It uses **ListField** to serialize the values of multiple categories: genre, director, actor, and year. Each **ListField** contains a child argument specifying that each item in the list should be a string (**CharField**).

The **required=False** parameter on each field indicates that these fields are optional. Users do not need to provide preferences for all categories; they can specify as many or as few preferences in each category as they like.

```

class WatchHistorySerializer(serializers.Serializer):
    title = serializers.CharField(max_length=255)
    year = serializers.IntegerField()
    director = serializers.CharField(max_length=255)
    genre = serializers.CharField(max_length=255)

```

The **WatchHistorySerializer** ensures that each field adheres to specified data types and constraints, such as character length limits for strings and integers for the year.

Let us create our service layer for adding preferences and watch history to the user, create a new file in **src/movies/services.py** and add the following service functions to it:

```

from typing import Dict, Any
from collections import defaultdict
from django.db import transaction
from django.contrib.auth import get_user_model
from django.shortcuts import get_object_or_404
from movies.models import UserMoviePreferences

def add_preference(user_id: int, new_preferences: Dict[str,
Any]) -> None:
    """

```

```

Adds new preferences or updates existing ones in the user's
movie preferences,
using defaultdict to automatically handle lists and avoiding
duplicate entries.
:param user_id: ID of the user
:param new_preferences: Dict containing new preferences to be
added or updated
"""
with transaction.atomic():
    user = get_object_or_404(get_user_model(), id=user_id)
    (
        user_preferences,
        created,
    ) =
    UserMoviePreferences.objects.select_for_update().get_or_create(
        user_id=user.id, defaults={"preferences": {}}
    )
    # Use defaultdict to automatically handle list creation
    # Convert existing preferences to defaultdict to ease
    updating
    current_preferences = defaultdict(list,
    user_preferences.preferences)
    for key, value in new_preferences.items():
        # Ensure value is not already in the list to avoid
        duplicates
        if value not in current_preferences[key]:
            current_preferences[key].append(value)

    # Convert defaultdict back to dict to ensure compatibility
    with Django models
    user_preferences.preferences = dict(current_preferences)
    user_preferences.save()

```

The **add\_preference** service function updates or adds new movie preferences for a user. Trapping database operations in a transaction ensures data integrity using **select\_for\_update** and atomic transaction. It uses **defaultdict** to simplify list management within the user's preferences, automatically create lists for new preference categories, and avoid duplicate

entries. After processing the latest preferences, it converts the **defaultdict** back to a regular dictionary to maintain compatibility with Django's model, then saves the updated preferences.

#### **INFO:**

***`select_for_update`** is a Django ORM query method used to lock selected rows against modification by other transactions until the current transaction is completed. When a query is executed with **`select_for_update()`**, the database locks the rows fetched by the query, ensuring that other transactions cannot change them until the lock is released.*

*The locking mechanism can increase wait times for other transactions trying to access the locked rows, potentially reducing the throughput of high-load systems.*

Now, in the same **src/movies/services.py** file, add the **add\_watch\_history** service function:

```
from typing import Dict
from django.shortcuts import get_object_or_404
from movies.models import Movie, UserMoviePreferences

def add_watch_history(user_id: int, movie_id: int) -> None:
    """
    Adds a new movie to the user's watch history.

    :param user_id: ID of the user
    :param movie_info: Dict containing information about the
        movie watched
    """
    movie = get_object_or_404(Movie, id=movie_id)
    movie_info = {
        "title": movie.title,
        "year": movie.release_year,
        "director": movie.extra_data.get("directors", []),
        "genre": movie.genres,
    }
    try:
        with transaction.atomic():
```

```

        user_preferences, created =
        UserMoviePreferences.objects.get_or_create(
            user_id=user_id, defaults={"watch_history": [movie_info]}
        )
    except IntegrityError:
        user_preferences =
        UserMoviePreferences.objects.get(user_id=user_id)
        created = False

    if not created:
        # Add new movie info to existing watch history
        current_watch_history = user_preferences.watch_history
        current_watch_history.append(movie_info)
        user_preferences.watch_history = current_watch_history
        user_preferences.save()

```

The **add\_watch\_history** service function updates a user's watch history with a new movie entry. It first checks if a record exists for the given **user\_id** and creates one with the initial **movie\_info** if it doesn't. For existing records, the new **movie\_info** is appended to the user's watch history list and saves the updated record. This method effectively handles adding movie watch events without duplicating the user record, but it supports adding the movie multiple times in the history.

Then, we need two functions to retrieve data for the read operations:

```

def user_preferences(user_id: int) -> Any:
    user_preferences = get_object_or_404(UserMoviePreferences,
        user_id=user_id)
    serializer =
    PreferencesSerializer(user_preferences.preferences)
    return serializer.data

def user_watch_history(user_id: int) -> dict[str, Any]:
    user_preferences = get_object_or_404(UserMoviePreferences,
        user_id=user_id)
    return {"watch_history": user_preferences.watch_history}

```

The **user\_preferences** function fetches and serializes the movie preferences for a user identified by **user\_id**. It retrieves the **UserMoviePreferences** model instance, serializes the preferences using **PreferencesSerializer** and returns the serialized data.

The **user\_watch\_history** function accesses a user's movie watch history using the **user\_id**. It finds the relevant **UserMoviePreferences** instance and returns the **watch\_history** in a dictionary format.

The new functions added to the service layer must be used whenever we want to add a user's watch history preferences. If another Django application uses the recommendation system, it should only import the **services.py** functions. Any management command and API views should follow this rule as a good practice.

Let us add the API views for preferences, open the file **src/movies/api.py** and add the new **UserPreferencesView** class with the POST and GET implementations:

```
from rest_framework.views import APIView
from rest_framework.request import Request
from rest_framework.response import Response
from rest_framework import status
from movies.serializers import AddPreferenceSerializer
from movies.services import add_preference, user_preferences

class UserPreferencesView(APIView):
    """
    View to add new user preferences and retrieve them.
    """

    def post(self, request: Request, user_id: int) -> Response:
        serializer = AddPreferenceSerializer(data=request.data)
        if serializer.is_valid():
            add_preference(user_id,
                           serializer.validated_data["new_preferences"])
            return Response(serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)

    def get(self, request: Request, user_id: int) -> Response:
        data = user_preferences(user_id)
        return Response(data)
```

The **UserPreferencesView** class is a RESTful API view that implements the POST and GET views for adding and getting user preferences. The POST

method uses the **AddPreferenceSerializer** for data validation. The code uses the **is\_valid** serializer method to verify if the payload is valid. When the payload is valid, the view calls the **add\_preference** service function to persist the new preferences and then returns a 201 status code. When the validation fails, the API returns a 400 Bad Request response with error details. The GET method retrieves a user's saved preferences from the database and uses the **PreferencesSerializer** for the response.

Now, let us add the watch history API view in the same file:

```
from rest_framework.views import APIView
from rest_framework.request import Request
from rest_framework.response import Response
from rest_framework import status
from movies.serializers import AddToWatchHistorySerializer
from movies.services import user_watch_history,
add_watch_history
class WatchHistoryView(APIView):
    """
    View to retrieve and add movies to the user's watch history.
    """

    def get(self, request: Request, user_id: int) -> Response:
        data = user_watch_history(user_id)
        return Response(data)

    def post(self, request: Request, user_id: int) -> Response:
        serializer = AddToWatchHistorySerializer(data=request.data)
        if serializer.is_valid():
            add_watch_history(
                user_id,
                serializer.validated_data["id"],
            )
            return Response(
                {"message": "Movie added to watch history."},
                status=status.HTTP_201_CREATED,
            )
        return Response(serializer.errors,
            status=status.HTTP_400_BAD_REQUEST)
```

The **RetrieveWatchHistoryView** class implements the GET for fetching the watch history and POST for updating it with new movies.

The GET method locates the user's watch history using the function service **user\_watch\_history** with the provided **user\_id**. The watch history is returned as part of the response payload if the user's preferences are found.

The POST method allows users to add movies to the user's watch history. It first validates the incoming request data using the **AddToWatchHistorySerializer**. When the data is valid, call the **add\_watch\_history** service function, passing the **user\_id** and **movie\_id**. A successful addition results in a 201 Created response. Conversely, if the data fails to pass validation, the API returns a 400 Bad Request response, including detailed error messages.

## [Understanding SPARQL](#)

The Resource Description Framework (RDF) is a robust and standardized model for data description and interchange on the Web. RDF represents information using a simple triple-based structure comprising a subject, predicate, and object.

SPARQL is a language designed specifically for querying databases that store information with the RDF. Let us see an example query to understand how to use this language:

```
SELECT ?movie ?movieLabel WHERE {  
  ?movie wdt:P31 wd:Q11424.  
  SERVICE wikibase:label { bd:serviceParam wikibase:language "  
    [AUTO_LANGUAGE],en". }  
}  
LIMIT 10
```

The subject in our query is the **?movie**, representing the movies we are trying to find. The predicate is **wdt:P31**, the wiki data code for the type of relationship “instance of.” The object is **wd:Q11424**, which in Wikidata is the “film.”

The **SERVICE wikibase:label** segment of the query is not part of the triple structure. It is a directive to the Wikidata Query Service to fetch the human-readable labels automatically.

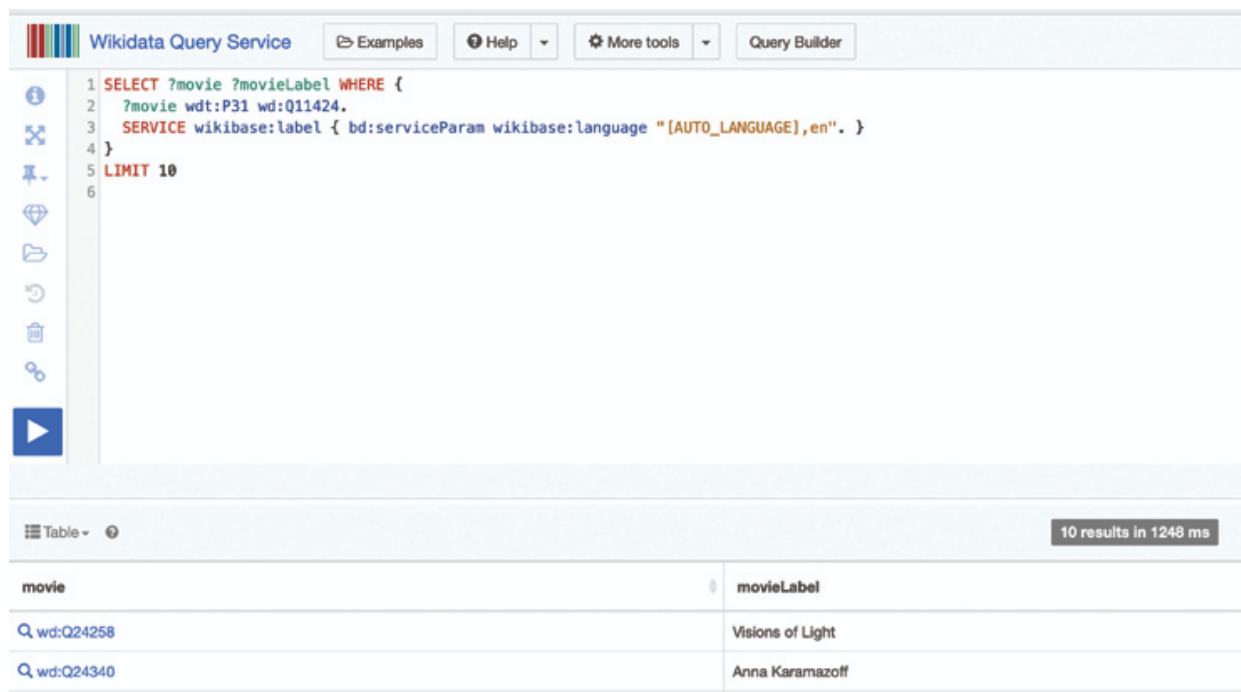


## INFO:

If you want to dive deep into SPARQL, you can check the official specifications and documentation for SPARQL at the following link <https://www.w3.org/TR/sparql11-overview/>

Suppose we were to translate the triple pattern `?movie wdt:P31 wd:Q11424` into a natural language statement, it might read as “Find entities (`?movie`) that are instances of (`wdt:P31`) the class ‘film’ (`wd:Q11424`).”

If you want to try the query, you can use the web interface of the SPARQL query service at <https://query.wikidata.org/>.



The screenshot shows the Wikidata Query Service web interface. At the top, there's a header with the Wikidata logo, the text "Wikidata Query Service", and buttons for "Examples", "Help", "More tools", and "Query Builder". Below the header, a SPARQL query is entered in a text area:

```
1 SELECT ?movie ?movieLabel WHERE {  
2   ?movie wdt:P31 wd:Q11424.  
3   SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }  
4 }  
5 LIMIT 10  
6
```

Below the query area, there's a "Table" view button and a status bar indicating "10 results in 1248 ms". The results are displayed in a table with two columns: "movie" and "movieLabel".

movie	movieLabel
<a href="#">Q24258</a>	Visions of Light
<a href="#">Q24340</a>	Anna Karamazoff

*Figure 3.2: Web interface Wikidata Query Service*

For our project, we access the query results via the HTTP protocol. Wikidata provides the following endpoint to get the results:

URL	METHOD	PARAMS	HEADERS
<a href="https://query.wikidata.org/sparql">https://query.wikidata.org/sparql</a>	GET	The endpoint accepts the query as a GET parameter with the name “query”	The accept headers of the request can use different formats. We will use the application/json which is a good fit for our example.

*Table 3.2: Endpoints*

Here is an example of how to get the JSON results using the Python requests library:

```
>> query = """
    SELECT ?movie ?movieLabel WHERE {
      ?movie wdt:P31 wd:Q11424.
      SERVICE wikibase:label { bd:serviceParam wikibase:language "
        [AUTO_LANGUAGE],en". }
    }
LIMIT 10
"""

>> url = "https://query.wikidata.org/sparql"
>> response = requests.get(url, headers={"Accept":
"application/json"}, params={"query": query})
>> print(response.json().keys())
dict_keys(['head', 'results'])
```

To provide a good user experience with our recommendation system, we need to obtain as much information as possible about the movies in our database.

The query we are going to use to retrieve the data from Wikidata is the following one:

```
SELECT ?film ?filmLabel
  (GROUP_CONCAT(DISTINCT ?genreLabel; separator=", ") AS ?
  genres)
  (GROUP_CONCAT(DISTINCT ?countryLabel; separator=", ") AS ?
  countries)
  (GROUP_CONCAT(DISTINCT ?directorLabel; separator=", ") AS ?
  directors)
WHERE {
  ?film wdt:P31 wd:Q11424;           # Instance of film
    wdt:P577 ?releaseDate;         # With a release date
  OPTIONAL { ?film wdt:P136 ?genre. # Has genre
    ?genre rdfs:label ?genreLabel.
    FILTER(LANG(?genreLabel) = "en") }
  OPTIONAL { ?film wdt:P495 ?country. # Country of origin
    ?country rdfs:label ?countryLabel.
```

```

    FILTER(LANG(?countryLabel) = "en") }
OPTIONAL { ?film wdt:P57 ?director. # Director
    ?director rdfs:label ?directorLabel.
    FILTER(LANG(?directorLabel) = "en") }
FILTER(YEAR(?releaseDate) = 2010) # Films released in 2010
SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
GROUP BY ?film ?filmLabel
LIMIT 10

```

This SPARQL query fetches details of films released in 2010, including their titles, genres, countries of origin, and directors. The **GROUP\_CONCAT** function aggregates multiple genres, countries, and directors for each film into single, comma-separated strings, ensuring a compact presentation of data. Using **OPTIONAL** blocks and English language filters guarantees the inclusion of films even when some data might be missing and provides the readability of results. By grouping the results by film, the query efficiently organizes the information, making it ideal for comprehensive analysis of films from that year within a concise output format. We filter by date to make it faster to retrieve the movies.

Here is a summary of Wikidata properties (PIDs) used in the query:

- **wdt:P31 (Instance of):** Identifies the specific class or type to which an item belongs. In the context of your query, it is used to specify items that are films.
- **wd:Q11424 (Film):** Represents the concept of a “film” or “movie” within Wikidata.
- **wdt:P577 (Publication Date):** Indicates when the film was first made available to the public, commonly used to filter films by their release year.
- **wdt:P136 (Genre):** Categorizes the film into one or more predefined genres, such as action, drama, or comedy, highlighting the thematic or stylistic characteristics of the film.
- **wdt:P495 (Country of Origin):** Specifies the country or countries where the film was produced, reflecting the geographical origins of the film’s production.

- **wdt:P57 (Director):** Points to the individual(s) who directed the film, emphasizing the key creative leadership role in the film's production.

## Getting the Data for Our Project

In the previous section, we learned how to retrieve data from the Wikidata query service. Now, we will create a script to get the data for our project. We will make a script that generates comma-separated values. Still, this script could be implemented as a management command or a background task, as we will see in [Chapter 4, Asynchronous Tasks and Data Processing](#). When working with a data science project, the data we need to process comes in different formats and CSV is a prevalent format.

The script will generate a CSV file per year that we will use in the next section to ingest data via the API. This script is divided into three sections for a clear and focused breakdown of its operations, each part highlighting specific functionalities and logic.

The first section imports the required Python modules and sets the base SPARQL query:

```
import csv
import requests
from datetime import datetime
from typing import Any
from tqdm import tqdm

# Base SPARQL query without the LIMIT and OFFSET
base_query = """
SELECT ?film ?filmLabel
  (GROUP_CONCAT(DISTINCT ?genreLabel; separator=", ") AS ?
   genres)
  (GROUP_CONCAT(DISTINCT ?countryLabel; separator=", ") AS ?
   countries)
  (GROUP_CONCAT(DISTINCT ?directorLabel; separator=", ") AS ?
   directors)
WHERE {
  ?film wdt:P31 wd:Q11424;
    wdt:P577 ?releaseDate;
  OPTIONAL { ?film wdt:P136 ?genre.
    ?genre rdfs:label ?genreLabel.
```

```

        FILTER(LANG(?genreLabel) = "en") }
    OPTIONAL { ?film wdt:P495 ?country.
        ?country rdfs:label ?countryLabel.
        FILTER(LANG(?countryLabel) = "en") }
    OPTIONAL { ?film wdt:P57 ?director.
        ?director rdfs:label ?directorLabel.
        FILTER(LANG(?directorLabel) = "en") }
    SERVICE wikibase:label { bd:serviceParam wikibase:language "
[AUTO_LANGUAGE],en". }
}
GROUP BY ?film ?filmLabel
"""

```

This section of the script imports necessary libraries for data handling and web requests, then defines a base SPARQL query to select movies from a dataset, grouping them by title and including grouped information on genres, countries of production, and directors. The query will filter by year and use pagination to ensure it will download all the data from datawiki.

The **tqdm** library in Python adds progress bars to loops, showing the progress, speed, and estimated completion time for lengthy operations.

```

def fetch_by_year(year: int) -> list[dict[str, Any]]:
    data = []
    limit = 1000 # Adjust as needed
    offset = 0
    while True:
        # Dynamically construct the query for each year
        query = base_query.replace("FILTER(YEAR(?releaseDate) = 2010)", f"FILTER(YEAR(?releaseDate) = {year})")
        query += f" LIMIT {limit} OFFSET {offset}"
        url = "https://query.wikidata.org/sparql"
        response = requests.get(
            url, headers={"Accept": "application/json"}, params=
            {"query": query}
        )
        if response.status_code == 200:
            batch_data = response.json()["results"]["bindings"]
            if not batch_data:

```

```

        break # Exit loop if no more data for this year
    data.extend(batch_data)
    offset += limit
else:
    print(f"Failed to retrieve data for {year}, status code:
    {response.status_code}")
    break # Exit loop in case of an HTTP error
return data

```

This function, **fetch\_by\_year**, queries a SPARQL endpoint for movies released in a specified year, returning the results as a list of dictionaries. It modifies a base query to filter movies by the provided year, then paginates through results in batches of 1000 (adjustable) by updating the offset after each successful request. If a request fails or no more data is available, the function exits and returns the collected data.

```

def fetch_all_data() -> None:
    current_year = datetime.now().year

    for year in tqdm(range(1888, current_year + 1),
total=current_year + 1 - 1888):
        data = fetch_by_year(year)

    # CSV file name adjusted for a comprehensive dataset
    csv_file_name = f"movies_data_{year}_to_current.csv"

    with open(csv_file_name, mode="w", newline='', encoding="utf-
8") as file:
        writer = csv.writer(file)
        writer.writerow(["title", "genres", "country", "extra_data"])
        for item in data:
            directors = item.get("directors", {}).get("value", "")
            writer.writerow([
                item.get("film", {}).get("value", ""),
                item.get("filmLabel", {}).get("value", ""),
                item.get("genres", {}).get("value", ""),
                item.get("countries", {}).get("value", ""),
                {"directors": directors},
            ])

if __name__ == "__main__":

```

```
fetch_all_data()
```

The `fetch_all_data` function iterates through every year from 1888 to the current year, fetching movie data for each year using the `fetch_by_year` function. It then writes this data into a CSV file named for the range of years processed, creating a comprehensive dataset. Each year includes details like the film's title, genres, countries of production, and directors. The progress of this operation is visually tracked using a `tqdm` progress bar, providing feedback on the completion status.

The script will generate several CSV files with the filename `movies_data_XXXX_to_current.csv` where XXXX refers to the year.

To use the script, we need to install the requests library for our development environment:

```
rye add --dev requests tqdm
rye sync
rye run python src/download_movies.py
```

When running the script you should see the progress of the download:

```
16%|███████████          | 31/137  
[08:56<43:17, 24.51s/it]
```

**INFO:**

You don't need to execute the script to download the movies. You can download it from the following link: <https://bit.ly/3TGvIDu>

Using the requests library, this Python script queries Wikidata to gather information on films, including their titles, genres, countries of origin, and directors, via SPARQL. Pagination allows us to retrieve the datasets, efficiently fetching data in manageable batches. As the script iterates through the data, the results accumulate until all relevant entries have been retrieved. Following data collection, the script uses the CSV module to organize and write the data into a CSV file named `"movies_data.csv"`. This process involves creating a structured format with headers for each film attribute, enabling easy analysis and review.

## Creating and Testing APIs for Data Ingestion

This section will create an endpoint to ingest data for our project. The new API will accept files in different formats and process and transform the data to persist in the database as a movie.

Attribute	Details
Endpoint	/api/v1/movies/upload
HTTP Method	POST
Content-Type	multipart/form-data
Payload	File data in the form of form-data under the key file.
Supported File Types	text/csv, application/json, application/xml, application/vnd.ms-excel
Response	JSON object containing details of the uploaded data or an error message.
Status Codes	201 Created on success, 400 Bad Request for validation errors

**Table 3.3:** *Different Attributes and Details*

Here is an example of usage with curl:

```
curl -X POST -F "file=@short_movies_data.csv;type=text/csv" -H
"Content-Type: multipart/form-data"
http://localhost:8000/api/movies/upload/
```

The response when the file was processed successfully will be:

```
{
  "status": "success",
  "message": "File uploaded and processed successfully.",
  "processed_records": 15
}
```

When there is an error, the API will return the following error response:

```
{
  "status": "error",
  "error": "Unsupported file type. Please upload a supported
format."
}
```

Using our API design, we are ready to create the tests. We test some edge cases to ensure we cover the minimal requirements of our API design.



```
import pytest
from django.urls import reverse
from django.core.files.uploadedfile import SimpleUploadedFile
from rest_framework.test import APIClient

test_data = [(
    "file.csv",
    "text/csv",
    b"title,genres,extra_data\ntest,comedy,{\"directors\":\n[\"name\"]}\n",
    201,
), # Expected to succeed for CSV
(
    "file.json",
    "application/json",
    b'[{\"title\": \"test\", \"genres\": [\"comedy\"], \"extra_data\":\n{\"directors\": [\"name\"]} }]',
    201,
), # Expected to succeed for JSON
(
    "file.txt",
    "text/plain",
    b"This is a test.",
    400,
), # Unsupported file type, expecting failure
]

@pytest.mark.parametrize("file_name, content_type,
file_content, expected_status", test_data)
@pytest.mark.django_db
def test_general_upload_view(client: APIClient, file_name: str,
content_type: str, file_content: str, expected_status: int):
    # Generate the URL dynamically using "reverse"
    url = reverse("movies:file-upload")

    # Create an in-memory uploaded file
    uploaded_file = SimpleUploadedFile(name=file_name,
content=file_content, content_type=content_type)

    # Make a POST request to the GeneralUploadView endpoint
```

```

response = client.post(url, {"file": uploaded_file},
format="multipart")

# Assert that the response status code matches the expected
status
assert response.status_code == expected_status

```

The preceding test tests file upload scenarios using a collection of test cases, each represented by a tuple that includes file name, content type, file content, and the expected HTTP response status. The tests cover successful uploads for supported formats like CSV and JSON, expecting a 201 status code, and a failure case for an unsupported format. The test simulates file uploads to the application's endpoint and asserts the response status, verifying the application's upload functionality behaves as expected under different conditions.

Next, open the file `src/movies/serializers.py` and add the new **GeneralFileUploadSerializer** class:

```

class GeneralFileUploadSerializer(serializers.Serializer):
    file = serializers.FileField()

    def validate_file(self, value: InMemoryUploadedFile) ->
    InMemoryUploadedFile:
        # Validate file size (e.g., 10MB limit)
        if value.size > 10485760:
            raise serializers.ValidationError("The file size exceeds
            the limit of 10MB.")

        # Validate file MIME type
        allowed_types = ["text/csv", "application/json",
            "application/xml"]
        if value.content_type not in allowed_types:
            raise serializers.ValidationError("Unsupported file type.")

        return value

```

The **GeneralFileUploadSerializer** class in the Django REST Framework is designed to validate file uploads. It includes a **FileField** for the file input and a custom **validate\_file** method for enforcing two main constraints: file size and MIME type. The method restricts file sizes to under 10MB to prevent server overload. It checks the file's MIME type against an allowed

list (**text/csv**, **application/json**, **application/xml**) to ensure only specific file formats are accepted. If a file fails these checks, a **ValidationError** is raised, indicating either an excessive file size or an unsupported file type.

Open the service file located at **src/movies/services.py** and add the new service class and functions to process files:

```
import csv
import json
from django.core.exceptions import ValidationError
from typing import Callable
from movies.utils import create_or_update_movie

def parse_csv(file_path: str) -> int:
    movies_processed = 0
    with open(file_path, encoding="utf-8") as file:
        reader = csv.DictReader(file)
        for row in reader:
            create_or_update_movie(**row)
            movies_processed += 1
    return movies_processed

def parse_json(file_path: str) -> int:
    movies_processed = 0
    with open(file_path, encoding="utf-8") as file:
        data = json.load(file)
        for item in data:
            create_or_update_movie(**item)
            movies_processed += 1
    return movies_processed

class FileProcessor:
    def process(self, file_path: str, file_type: str) -> int:
        if file_type == "text/csv":
            movies_processed = parse_csv(file_path)
        elif file_type == "application/json":
            movies_processed = parse_json(file_path)
        else:
            raise ValidationError("Invalid file type")

        return movies_processed
```

The **parse\_csv** function processes CSV files containing movie data. Using **CSV.DictReader**, it iterates over each row and updates or adds movie records through **create\_or\_update\_movie**. It counts and returns the total number of movies processed, facilitating efficient bulk updates from CSV sources.

**parse\_json** works similarly, but for JSON files, the file is loaded to parse an array of movie objects. Each is processed to update or add to the database, with the function returning the count of movies processed, enabling easy integration of movie data from JSON formats.

**FileProcessor** chooses between CSV and JSON parsers based on file type. It uses **process\_file** to call the appropriate parser, handling different data formats flexibly. The class ensures file type validity, simplifies the extension to additional formats, and enhances data pipeline integration. The class **FileProcessor** can be easily extended to support more formats.

Now, we will add the new API, open the file **src/movies/api.py** and add the new **GeneralUploadView**:

```
from contextlib import contextmanager
from django.core.files.storage import default_storage
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from typing import Any
from movies.serializers import GeneralFileUploadSerializer
from movies.processors import FileProcessor

@contextmanager
def temporary_file(uploaded_file):
    try:
        file_name = default_storage.save(uploaded_file.name,
        uploaded_file)
        file_path = default_storage.path(file_name)
        yield file_path
    finally:
        default_storage.delete(file_name)

class GeneralUploadView(APIView):
    def post(self, request, *args: Any, **kwargs: Any) ->
    Response:
```

```

serializer = GeneralFileUploadSerializer(data=request.data)
if serializer.is_valid():
    uploaded_file = serializer.validated_data["file"]
    file_type = uploaded_file.content_type

    with temporary_file(uploaded_file) as file_path:
        processor = FileProcessor()
        movies_processed = processor.process(file_path, file_type)
        return Response(
            {
                "message": f"{movies_processed} movies processed
                successfully."
            },
            status=status.HTTP_201_CREATED,
        )
else:
    return Response(serializer.errors,
                    status=status.HTTP_400_BAD_REQUEST)

```

The **temporary\_file** context manager safely handles an uploaded file by temporarily saving it, providing its file path for processing, and then automatically deleting it afterward. This ensures the temporary use of server resources and maintains data security by removing the file once it is done.

#### **INFO:**

*A context manager in Python automates the setup and teardown of resources, ensuring they are handled adequately around a block of code. Used with the statement, it manages resource allocation and release, like opening and closing files or managing database connections, to prevent resource leaks and simplify code for resource management.*

*A context manager uses the **\_\_enter\_\_** method to initialize resources at the start of a block and the **\_\_exit\_\_** method to clean up resources when exiting.*

The **GeneralUploadView** class implements the API view for handling file uploads. It validates uploaded files using **GeneralFileUploadSerializer**, manages temporary storage with **temporary\_file**, and processes files based on their type via **FileProcessor**. Successful processing returns the count of

processed items, while validation errors trigger appropriate error responses. This design ensures efficient file handling within a secure and maintainable framework.

### INFO:

*Django-Extensions is a third-party collection of custom extensions for the Django Framework. It includes a set of additional management commands, model fields, and other utilities that aim to enhance the development experience with Django.*

*You can install it using rye:*

```
rye add django-extensions
rye sync
```

*Open the project settings at **src/recommendation\_system/settings.py** and add django-extensions app:*

```
INSTALLED_APPS = [
    ...
    "django_extensions",
    ...
]
```

*Then you will have many additional features, like listing the URLs:*

```
rye run python src/manage.py show_urls
```

We still need to add the service layer function to create or update the movies, open the file **src/movies/services.py** and add the new function:

```
from typing import Tuple
from .models import Movie
from django.core.exceptions import ValidationError
import datetime

def create_or_update_movie(
    title: str,
    genres: list,
    country: str | None = None,
    extra_data: dict[Any, Any] | None = None,
    release_year: int | None = None
) -> Tuple[Movie, bool]:
```

```

"""
    Service function to create or update a Movie instance.
"""
try:
    # Ensure the release_year is within an acceptable range
    current_year = datetime.datetime.now().year
    if release_year is not None and (release_year < 1888 or
    release_year > current_year):
        raise ValidationError("The release year must be between
        1888 and the current year.")

    # Attempt to update an existing movie or create a new one
    movie, created = Movie.objects.update_or_create(
        title=title,
        defaults={
            "genres": genres,
            "country": country,
            "extra_data": extra_data,
            "release_year": release_year,
        }
    )
    return movie, created
except Exception as e:
    raise ValidationError(f"Failed to create or update the movie:
    {str(e)}")

```

The **create\_or\_update\_movie** function is designed to efficiently manage movie records in a database, determining whether to create a new entry or update an existing one based on the movie's title. It accepts details such as the movie's title, genre, country, directors, and release year. The function checks if the release year falls within a valid range from 1888 to the current year to ensure historical accuracy and relevance. It uses Django's **update\_or\_create** method, which either updates an existing movie with the given title or creates a new record if no match is found. In case of any errors during the database operation, a **ValidationError** is raised.

Finally, we need to update the **src/movies/urls.py** to add the new file upload API:

```
urlpatterns = [
```

```

path("", MovieListCreateAPIView.as_view(), name="movie-api"),
path("<int:pk>/", MovieDetailAPIView.as_view(), name="movie-
api-detail"),
path(
    "user/<int:user_id>/preferences/",
    UserPreferencesView.as_view(),
    name="user-preferences",
),
path(
    "user/<int:user_id>/watch-history/",
    WatchHistoryView.as_view(),
    name="user-watch-history",
),
path("upload/", GeneralUploadView.as_view(), name="file-
upload"),
]

```

Now, we are ready to test our upload endpoint using curl:

```

$ curl -X POST -F
"file=@/Users/mandarina/workspace/django_rest_book/all_movies_d
ata.csv;type=text/csv" -H "Content-Type: multipart/form-data"
http://localhost:8000/api/v1/movies/upload/
{"message":"1146 movies processed successfully."}%

```

As you will see, processing the **all\_movies\_data.csv** file will take several minutes. We still have several refinements to make in our project to ensure it is ready for production.

## Conclusion

We have detailed the development of data models and APIs necessary for a movie recommendation system, showing how to use Wikidata and SPARQL for data retrieval. The process outlines the importance of understanding user preferences and integrating structured data to offer personalized movie suggestions. This work sets the stage for building our movie recommendation system.

In the next chapter, we will refactor our API to implement asynchronous tasks, enabling our project to handle large-scale data processing.



## Questions

1. How does the movie model in this chapter extend to better suit the needs of a movie recommendation system?
2. Explain the concept of the “paradox of choice” in digital media platforms and its relevance to movie recommendation systems.
3. What are the benefits of using JSON fields in Django models for data science applications, particularly in the context of movie recommendations?

## Exercises

1. Add unit tests for **AddPreferenceSerializer** and **AddToWatchHistorySerializer**
2. Add tests for the service layer, as well as tests for the **add\_preference** and **add\_watch\_history** functions.
3. Extend the data ingestion to support XML data.
4. Write a Python script to fetch movie data using SPARQL and Wikidata.
  - a. The script should execute a SPARQL query to retrieve movies released in a specific year, including their genres and directors.
5. Extend the download script to resume the download when something goes wrong.
6. Create a script to merge all the CSV files into one.

# **CHAPTER 4**

## **Asynchronous Tasks and Data Processing**

### **Introduction**

A vast amount of data is expected to be processed when dealing with a data analytics project. We need a solution that has more real-world usage than the solution we implemented in the previous chapter. In this chapter, we will iterate on our project architecture and refactor it many times to improve its scalability by using asynchronous tasks. We will learn new concepts in each iteration to make our application more scalable.

### **Structure**

In this chapter, we will discuss the following topics:

- Introduction to Background Task Processing
- Introduction to Celery
- Celery Local Development Environment
- Example Task with Celery
- Background Tasks with Celery and Django
- Testing Asynchronous Data Processing
- Refactor API for Background Task Processing
- Using Cloud File Storage
- Handling Large Data Sets and Long-Running Processes

### **Introduction to Background Task Processing**

The backend constructed in the previous chapter is synchronous. When a request is made, the server must complete processing the current request before proceeding to the next one. In our API context, a synchronous

backend significantly limits scalability, particularly when handling large datasets or complex processing tasks.

Background task processing is a fundamental concept in software architecture that allows long-running tasks to be handled separately from the main application flow. This pattern enhances software applications' scalability, responsiveness, and user experience.

**INFO:**

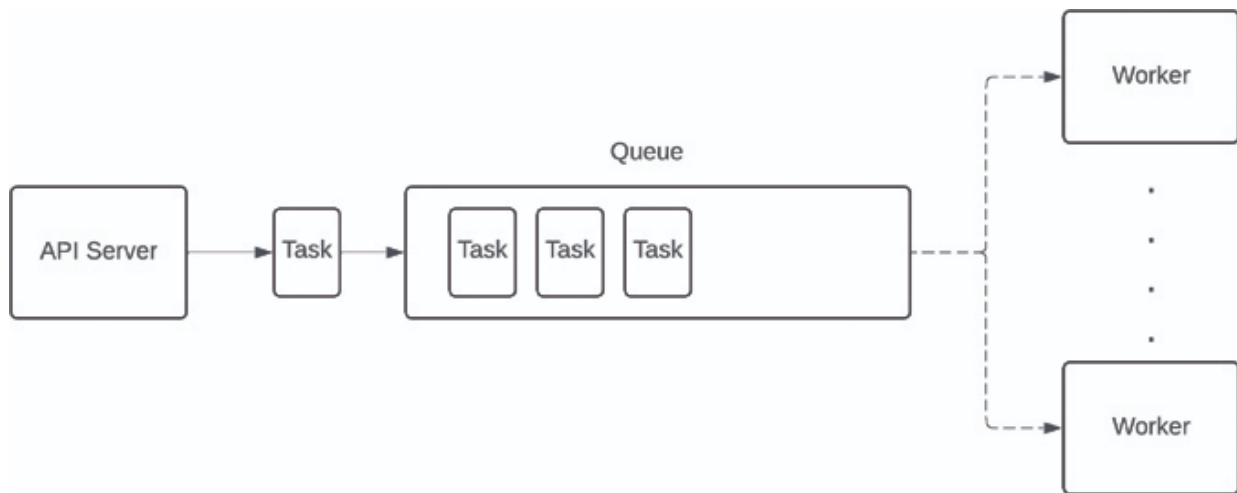
*In the previous chapter, we used the Django development server, which is unsuitable for running the backend in production due to its inability to handle multiple concurrent requests efficiently and its lack of integration with WSGI servers needed for high-performance applications.*

*In [Chapter 8, Deploying Your Data Science API](#), we will see how to use gunicorn to deploy in a way that's efficient, scalable, and suitable for handling multiple requests simultaneously. In the context of synchronous backend and gunicorn, the problem still needs to be solved since Gunicorn spawns numerous worker processes, each capable of handling one request at a time.*

*Gunicorn (Green Unicorn) is a Python WSGI HTTP Server for UNIX. It is widely used to serve Python web applications from various frameworks, such as Flask, Django, and others.*

In background task processing, several components work together to implement the pattern. Let us list these components and then review how they interact with each other:

- **API Server:** The Django server that provides the API endpoints.
- **Task Queue:** A queue system that allows for tasks to be enqueued and provides a mechanism for consumers to dequeue the work. The queue should also track and prevent the same work from being processed multiple times.
- **Worker:** A service that has the logic to process the tasks.



*Figure 4.1: Background task processing architecture*

When a request is made to the API server, it will be added to a queue instead of being processed immediately. The task queue is a buffer that temporarily stores tasks before processing. It decouples the task submission from execution, enabling asynchronous processing.

Workers will pick up tasks from the queue and start the execution. This will allow the API server to offload work to one or more workers that can be easily scaled. The workers can be implemented with any technology if they know how to dequeue tasks.

## Introduction to Celery

Celery is a widely adopted solution for implementing background tasks with Python. Celery is an open-source, distributed task queue system written in Python, designed to handle asynchronous task execution and scheduling in a distributed environment.

Here are some of the top features that make Celery a good choice for background task processing:

- **Distributed Task Processing:** Celery allows for executing tasks across multiple workers and machines, enabling applications to scale horizontally and manage increased workloads efficiently.
- **Broker Support:** It supports a variety of message brokers, including RabbitMQ, Redis, and Amazon SQS.

- **Task Scheduling and Periodic Tasks:** Celery can schedule tasks to run at specific times or intervals, similar to cron jobs using the Celery Beat scheduler.
- **Task Chaining and Workflows:** This feature allows complex workflows to be chained or grouped. This enables the sequential execution of tasks, where the output of one task can be used as input for the next.
- **Result Backend:** Celery can store or cache task results using backends like databases (for example, MySQL, PostgreSQL), cache frameworks (for example, Memcached, Redis), or message queues, allowing for easy retrieval and use of task outcomes.

**INFO:**

*Celery requires a message broker to route messages between the main application and workers executing the tasks.*

*The message broker acts as a middleman, facilitating message exchanges between Celery's task producers (your application) and consumers (worker processes).*

## **Celery Local Development Environment**

Before running a minimal celery sample task, we need to set up our local environment. Our local environment will use docker and docker-compose to set up the services required to run the message broker.

Docker is an open-source platform that simplifies the process of building, running, sharing, and managing containers. Containers allow developers to package an application with all its dependencies into a standardized unit for software development. This encapsulation makes it possible for an application to run in any environment, regardless of any customized settings that might differ from the environment in which the application was developed or tested. Docker provides the tools and workflows to manage the lifecycle of containers, including development, deployment, and scaling. By isolating applications into separate containers, Docker ensures quick, reliable, and consistent deployments, regardless of the target environment, whether on a local machine, in the cloud, or a hybrid setup.

Docker Compose is a tool used to define and run multi-container Docker applications. With Compose, you use a YAML file to configure your application's services, networks, and volumes. Then, you can create and start all the services from your configuration with a single command. This simplifies orchestrating multiple containers that make up an application, managing them as a single service. For instance, a web application that requires a database, a web server, and a reverse proxy could run each component in a separate container, with Docker Compose managing the networking and dependencies between them. Docker Compose thus improves the Docker experience, making it more efficient and manageable for development and testing.

The first requirement is to install Docker. Follow the official installation documentation to install the latest version (<https://docs.docker.com/engine/install/>).

Once you have installed docker, you must install the compose plugin by following the official steps provided here: <https://docs.docker.com/compose/install/linux/#install-the-plugin-manually>.

To simplify it, we will start a docker-compose configuration file that will start a Redis service.

We will extend this docker-compose configuration file in the following sections and chapters with more services.

Create a new file with the name **docker-compose.yml** in the root directory of the project:

```
version: '3.8'
services:
  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
```

This Docker Compose configuration defines a service named **redis**, using the lightweight **redis:alpine** image from Docker Hub, based on the Alpine Linux distribution. The ports directive maps port **6379** on the host to the same port inside the Redis container, enabling applications on the host to connect via **localhost:6379** or **127.0.0.1:6379**.

Run `docker-compose` to start your application. This command reads the `docker-compose.yml` file in the current directory and starts the services defined in it. By default, it runs in the foreground, providing logs to the console.

Let us start the services with the following command:

```
docker compose up
```

## Example Task with Celery

Let us build a simple example that explains how celery works. For this example, we will create a task that simulates a long-running operation. We will use Redis as the message broker.

First, install the dependencies using `rye`:

```
rye add redis celery
rye sync
```

Create a new file in `examples/celery_app.py` with the following task implementation:

```
from celery import Celery

# Create a Celery instance and configure it to use Redis as the
# message broker
app = Celery("example", broker="redis://localhost:6379/0")

# Define a task
@app.task
def long_running_task(x):
    import time
    time.sleep(15) # Simulate a long-running operation
    return x * x
```

The `celery_app.py` script sets up a basic Celery configuration to handle asynchronous tasks in Python. It defines a Celery application named 'example' that uses Redis as a message broker. A task, `long_running_task`, is created to simulate a time-consuming operation by sleeping for 15 seconds before returning the square of its input. This setup demonstrates how to offload tasks that can be executed independently.

Let us start one celery worker with the celery command:

```
cd examples
```

```
rye run celery -A celery_app worker --loglevel=info
```

If everything is working as expected, we should see celery's output as follows:

A screenshot of a terminal window with a dark background. At the top, the command '\$ rye run celery -A celery\_app worker --loglevel=info' is entered. Below it is a green progress bar. The terminal then displays the Celery version 'celery@mandarinas-MBP v5.3.6 (emerald-rush)' and the system information 'macOS-14.4.1-arm64-arm-64bit 2024-04-03 18:59:40'. It shows configuration details for the app, transport (redis://localhost:6379/0), results (disabled), concurrency (8), and task events (OFF). It also shows the queue configuration for 'celery' with exchange='celery(direct)' and key='celery'. Finally, it lists the task 'celery\_app.long\_running\_task' under the [tasks] section.

```
$ rye run celery -A celery_app worker --loglevel=info

celery@mandarinas-MBP v5.3.6 (emerald-rush)
macOS-14.4.1-arm64-arm-64bit 2024-04-03 18:59:40

[config]
.> app:          example:0x1020d0710
.> transport:    redis://localhost:6379/0
.> results:      disabled://
.> concurrency:  8 (prefork)
.> task events:  OFF (enable -E to monitor tasks in this worker)

[queues]
.> celery        exchange=celery(direct) key=celery

[tasks]
. celery_app.long_running_task
```

*Figure 4.2: Celery's output*

Next, we must enqueue tasks so the worker can start processing the work. Open a Python terminal to enqueue some work:

```
cd examples
rye run python
>>> from celery_app import long_running_task
>>> result = long_running_task.delay(4)
>>> print('Task enqueued:', result.id)
Task enqueued: 1d0a12b7-f9ff-4b54-ad28-78ca4c289430
```

If you open the celery terminal, you should see some logging activity as follows:

```
[2024-04-03 17:57:55,563: INFO/MainProcess] Task
celery_app.long_running_task[1d0a12b7-f9ff-4b54-ad28-
78ca4c289430] received
[2024-04-03 17:58:00,570: INFO/ForkPoolWorker-8] Task
celery_app.long_running_task[1d0a12b7-f9ff-4b54-ad28-
78ca4c289430] succeeded in 5.003338209004141s: 16
```

## [Background Tasks with Celery and Django](#)



Configuring Celery with Django is easy and requires a few steps. First, we must create a new **celery.py** file in the directory where the **settings.py** exists. The directory where we need to create the file is **src/recommendation\_system/celery.py**. Add the following code to configure Celery:

```
import os
from celery import Celery

os.environ.setdefault("DJANGO_SETTINGS_MODULE",
    "recommendation_system.settings")

app = Celery("movies")
app.config_from_object("django.conf:settings",
    namespace="CELERY")
app.autodiscover_tasks()
```

The code snippet integrates Celery with a Django project to manage background tasks. With **os.environ.setdefault**, initializes a Celery app with the project name, configures it with Django's settings using **config\_from\_object**, and finally, discovers and registers tasks within the project automatically using **autodiscover\_tasks()**. This setup enables background task processing in a Django application.

We need to ensure that the app is loaded when Django starts so that the **@shared\_task** decorator works. Open the file **src/recommendation\_system/\_\_init\_\_.py** and add the following code:

```
# This will make sure the app is always imported when
# Django starts so that shared_task will use this app.
from .celery import app as celery_app

__all__ = ("celery_app",)
```

Next, open the **src/recommendation\_system/settings.py** and add the **CELERY\_BROKER\_URL**:

```
CELERY_BROKER_URL = os.getenv("CELERY_BROKER_URL",
    "redis://localhost:6379/0")
```

The new setting uses **os.getenv** to prioritize the environment variable **CELERY\_BROKER\_URL**; if it is not set, it defaults to localhost.

We are now ready to create our first task for our Django project, create a new file in **src/movies/tasks.py** and add the new task to process files:

```

from celery import shared_task
from movies.services import FileProcessor

@shared_task
def process_file(filename: str, file_type: str) -> int:
    processor = FileProcessor()
    return processor.process(filename, file_type)

```

This code snippet defines a Celery shared task named **process\_file** which is designed to process a file given its **filename** and **file\_type**. The task leverages a service layer, precisely an instance of **FileProcessor** from the **movies.services** module to carry out the processing logic. A service layer is a beneficial design pattern as it encapsulates the business logic away from the application's view layer or task queue logic. This encapsulation allows for a clean separation of concerns, making the code more modular, easier to maintain, and testable. This architecture enhances code maintainability and improves the application's scalability by enabling the efficient distribution of tasks across worker processes.

To run the celery worker, open a terminal and execute:

```
rye run celery -A recommendation_system worker --loglevel=info
```

A results backend in Celery is a storage system that saves and retrieves the outcomes of asynchronous tasks. It allows for the centralized management of task results, supporting databases, message brokers, and caches as storage options. This enables clients to access task outcomes long after the tasks have been completed, which is helpful for processing or monitoring purposes.

In your settings.py, add the following configuration:

```

CELERY_RESULT_BACKEND = os.getenv("CELERY_RESULT_BACKEND",
    "redis://localhost:6379/0")

```

This line of code sets the **CELERY\_RESULT\_BACKEND** configuration option to the value of the **CELERY\_RESULT\_BACKEND** environment variable if it exists or defaults to **"redis://localhost:6379/0"**.

## [Testing Asynchronous Data Processing](#)

The following section will refactor the upload endpoint to support background workers. Celery allows you to change the project settings when

running tests to make the calls to `.delay()` and `.apply_async()` execute immediately and propagate exceptions if any occur.

We are going to create a new settings file for the tests. This file is an excellent way to maintain a separate configuration for testing, especially when adjusting settings that differ from other environments.

Create the `test_settings.py` file in the `src/recommendation_system/test_settings.py`. This file will override or extend your base settings specifically for tests.

The file will import everything from the settings file and then we can override specific settings.

```
from .settings import *  
  
CELERY_TASK_ALWAYS_EAGER = True  
CELERY_TASK_EAGER_PROPAGATES = True
```

When you run your tests, you must tell Django to use `test_settings.py` instead of the default `settings.py`. When invoking your tests, you can do this by setting the `DJANGO_SETTINGS_MODULE` environment variable. For example,

```
DJANGO_SETTINGS_MODULE=recommendation_system.test_settings rye  
run pytest
```

### INFO:

*Managing sensitive information carefully to avoid security risks is crucial when working with multiple configuration files in a Django project, such as `settings.py` for production and `test_settings.py` for testing. This is particularly important if your project is stored in a version control system (VCS) like Git.*

*Sensitive information might include API keys, database credentials, secret keys used for cryptographic signing, or any other data that could be exploited if it fell into the wrong hands. Such exposure could lead to unauthorized access, data breaches, or security vulnerabilities in your testing environment and production system.*

*To mitigate these risks, consider the following best practices:*

**Environment Variables:** *Instead of hardcoding sensitive data in your `test_settings.py`, use environment variables to inject these values at runtime. This approach keeps credentials from your source code and lets*

*you maintain different secrets for development, testing, and production environments without changing your code.*

***.env Files:** You can use .env files to define environment variables for local development and testing. Libraries like **django-environ** can help manage these variables within your Django project. Ensure **.env** files are added to your **.gitignore** to prevent them from being committed to your VCS.*

## **Refactor API for Background Task Processing**

The current implementation of the **GeneralUploadView** calls the service layer directly from the view. The resource-intensive service layer contains all the logic to process the uploaded files. We need to move the call of the service layer to a worker and queue the file. The first iteration of the refactor will still use temporary files; however, this approach will not work in a distributed system. By the end of this chapter, we will have a working solution for a distributed system.

Open the file **src/movies/api.py** and replace the implementation of **GeneralUploadView** with the new one:

```
from movies.tasks import process_file

class GeneralUploadView(APIView):
    def post(self, request, *args: Any, **kwargs: Any) ->
    Response:
        serializer = GeneralFileUploadSerializer(data=request.data)
        if serializer.is_valid():
            uploaded_file = serializer.validated_data["file"]
            file_type = uploaded_file.content_type

            with temporary_file(uploaded_file) as file_path:
                # Celery call using delay
                process_file.delay(file_path, file_type)
            return Response(
                {"message": f"Your file is being processed."},
                status=status.HTTP_202_ACCEPTED,
            )
        else:
            return Response(serializer.errors,
                status=status.HTTP_400_BAD_REQUEST)
```

The new implementation closely mirrors the former, with the main change being the shift from calling the service class **FileProcessor** to calling the task **process\_file** directly, which now queues the **filename** and **file\_type**. Due to size constraints, we are not sending the file contents, as storing them in Redis (the message broker) would be impractical.

Also, the status code and the message were changed to inform our users that the file is being processed.

**INFO:**

*The HTTP 202 Accepted status code indicates that a request has been received and understood but has yet to be acted upon. It is commonly used in scenarios where a request is processed asynchronously, meaning the server has accepted the request for processing but still needs to complete it. The 202 status is particularly useful in APIs and web services for tasks requiring significant processing time.*

If we try to execute the tests with rye, it will fail:

```
DJANGO_SETTINGS_MODULE=recommendation_system.test_settings rye
run pytest -vvv
===== 2 failed, 25 passed in 1.85s
=====
```

Tests are failing due to a change in the status code to the new 202. This change is breaking the API contract. Since our first iteration of the upload API was a prototype, we will change it; however, if there are already API users, this change should not be made.

Let us update the test data to use the new status code, open the file **src/movies/tests/test\_api.py** and change **test\_data** to the new one:

```
test_data = [
    (
        "file.csv",
        "text/csv",
        b'title,genres,extra_data\ntest,comedy,{"directors":'
        '[{"name"}]\n',
        202,
    ), # Expected to succeed for CSV
    (
```

```

        "file.json",
        "application/json",
        b'[{ "title": "test", "genres": ["comedy"], "extra_data":
        {"directors": ["name"]}]',
        202,
    ), # Expected to succeed for JSON
    (
        "file.txt",
        "text/plain",
        b"This is a test.",
        400,
    ), # Unsupported file type, expecting failure
]

```

If we execute the test again, we should get green pass tests:

```

rye run pytest -vvv
===== 27 passed in 1.74s
=====

```

Test passed!

When your application is distributed across multiple machines or using worker processes that may run on different instances, using local filesystems for file processing becomes impractical. This is because the machine that handles the file upload is not guaranteed to be the same machine that processes the file, leading to issues where the file is inaccessible across different instances. A standard solution to this problem is to use cloud storage services, such as Amazon S3, which provide a centralized, accessible-anywhere storage solution. In the next section, we will refactor the current solution to use cloud storage.

## [Using Cloud File Storage](#)

Today, it is more than typical for our services to be deployed in Kubernetes or similar environments where one process could be executed on a different server or node. We need to refactor our upload API again to support cloud file storage, which will make our uploaded files available to any server or node.

Several providers offer file storage. To make our code as agnostic as possible, we will use **django-storages**. Django Storages is a collection of custom storage backends for Django that are designed to allow you to integrate your Django projects with different storage services like Amazon S3, Google Cloud Storage, Microsoft Azure Storage, and many others. It abstracts the complexities of dealing with these services directly, allowing developers to work with files uniformly, regardless of where they are stored.

We will opt for S3 since it is one of the most popular. However, if you have to use a different provider with **django-storages**, you will only need to change the project's settings. The code of our API will not be tightly coupled to a specific solution.

First, open a terminal and install **django-storages** using rye:

```
rye add django-storages\[boto3\]
rye sync
```

Now we need to add storages to our **INSTALLED\_APPS**, open the file **src/recommendation\_system/settings.py** and add it to the list:

```
INSTALLED_APPS = [
    ...
    "storages",
    ...
]
```

Now, create a new file with the production settings in the **src/recommendation\_system/production\_settings.py** directory with the following contents:

```
import os
from settings import *
DEFAULT_FILE_STORAGE =
"storages.backends.s3boto3.S3Boto3Storage"
AWS_ACCESS_KEY_ID = os.environ["AWS_ACCESS_KEY_ID"]
AWS_SECRET_ACCESS_KEY = os.environ["AWS_SECRET_ACCESS_KEY"]
AWS_STORAGE_BUCKET_NAME = "test-bucket"
```

The configuration **DEFAULT\_FILE\_STORAGE** sets the storage configuration to use AWS S3. If you want to use a different provider like Google, change it to **storages.backends.gcloud.GoogleCloudStorage**.

Next, we have the AWS access key and secret key, which read values from the environment variables to prevent any configuration from being committed to the code repository. Finally, we have the bucket name for putting or getting the S3 objects.

**INFO:**

*AWS S3 (Simple Storage Service) is a scalable, high-speed, web-based cloud storage service designed for online backup and archiving of data and application programs. It provides a simple interface that allows users to store and retrieve any amount of data, anytime, from anywhere on the web. S3 is known for its durability, availability, and scalability, making it an ideal choice for various applications, from websites and mobile apps to backup, recovery, and more. It supports data transfer over SSL and automatic data encryption once stored. S3 operates on a pay-per-use model, offering cost-effective storage solutions to businesses of all sizes.*

Changes for the upload endpoint are minimal. Let us see the new version:

```
import os
import uuid

from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .serializers import GeneralFileUploadSerializer
from django.core.files.storage import default_storage
from django.core.files.base import ContentFile

class GeneralUploadView(APIView):
    def post(self, request, *args: Any, **kwargs: Any) ->
    Response:
        serializer = GeneralFileUploadSerializer(data=request.data)
        if serializer.is_valid():
            uploaded_file = serializer.validated_data["file"]
            file_type = uploaded_file.content_type

            # Extract the file extension
            file_extension = os.path.splitext(uploaded_file.name)[1]
            # Generate a unique file name using UUID
            unique_file_name = f"{uuid.uuid4()}{file_extension}"
```



```

# Save the file directly to the default storage
file_name = default_storage.save(unique_file_name,
ContentFile(uploaded_file.read()))
process_file.delay(file_name, file_type)

return Response(
    {"message": f"Job queued for processing."},
    status=status.HTTP_202_ACCEPTED,
)
else:
    return Response(serializer.errors,
        status=status.HTTP_400_BAD_REQUEST)

```

In the new version, the temporary file context manager was removed, and instead, we use Django's **default\_storage**. **default\_storage** is an abstraction that allows you to interact consistently with your project's default file storage system, regardless of where files are stored. The library **django-storages** provides a collection of custom storage backends for Django, supporting various solutions beyond Django's default local filesystem storage.

To prevent any potential overwriting, the filename uses **uuid**, guaranteeing a unique filename in the bucket.

Let us see the new changes to the service layer:

```

class FileProcessor:
    def process(self, file_name: str, file_type: str) -> int:
        # Check if the file exists in the default storage
        if default_storage.exists(file_name):
            # Open the file directly from storage
            with default_storage.open(file_name, "r") as file:
                if file_type == "text/csv":
                    movies_processed = parse_csv(file)
                elif file_type == "application/json":
                    movies_processed = parse_json(file)
                else:
                    raise ValidationError("Invalid file type")
            return movies_processed
        else:
            raise ValidationError("File does not exist in storage.")

```

The updated code introduces significant enhancements to the original **FileProcessor** class, transitioning from processing files located on the local filesystem to handling files stored on Amazon S3 using Django's storage system. Given the operations performed in **parse\_csv** and **parse\_json** expect a file path, we need to change the implementation to support a file object instead.

```
def parse_csv(file: IO[Any]) -> int:
    movies_processed = 0
    reader = csv.DictReader(file)
    for row in reader:
        create_or_update_movie(**row)
        movies_processed += 1
    return movies_processed

def parse_json(file: IO[Any]) -> int:
    movies_processed = 0
    data = json.load(file)
    for item in data:
        create_or_update_movie(**item)
        movies_processed += 1
    return movies_processed
```

The new functions now accept a file object and use it directly to open it using the **DictReader** or **json.load**.

With this new implementation, our workers could be executed at any server or node, and the tasks worker can process the files thanks to the cloud storage.

To run the tests, we need a new service that will mock S3 since we don't want to use a genuine S3 service when we execute the tests.

**INFO:**

*MinIO is an open-source, S3-compatible object storage service optimized for high performance and scalability. It's designed to store unstructured data like photos, videos, and backups. MinIO can easily integrate into development and production workflows, including Docker Compose environments. Its simplicity and compatibility with Amazon S3 APIs make it versatile for various applications, from web apps to machine learning projects.*

Open the **docker-compose.yml** file at the root directory and add the new **minio** service:

```
minio:
  image: minio/minio
  volumes:
    - minio_data:/data
  ports:
    - "9000:9000"
    - "9001:9001"
  command: server /data --console-address ":9001"
  environment:
    MINIO_ACCESS_KEY: minioadmin
    MINIO_SECRET_KEY: minioadmin
```

The preceding configuration defines a Docker Compose service for running a MinIO container. The **image** field specifies the MinIO Docker image to use. The volumes section mounts a named Docker volume, **minio\_data**, to **/data** inside the container. The ports section maps port **9000** on the host to port **9000** inside the container, which is used for accessing the MinIO API and S3-compatible services. Additionally, port **9001** on the host is mapped to port **9001** inside the container, designated for accessing MinIO's built-in web-based management console. The command field instructs the container to start the MinIO server with **/data** as its storage directory and specifies the console's address on port 9001. The environment section sets environment variables inside the container for the MinIO access and secret keys (**MINIO\_ACCESS\_KEY** and **MINIO\_SECRET\_KEY**), using **minioadmin** as the default credentials.

Still, we need to configure our **minio** instance to be able to run the tests. We need to add a **minio-setup** service that will clean up and create a bucket:

```
minio-setup:
  image: minio/mc
  depends_on:
    - minio
  entrypoint: >
    /bin/sh -c "
    /usr/bin/mc config host add myminio http://minio:9000
    minioadmin minioadmin;
```

```
/usr/bin/mc rm -r --force myminio/test-bucket;  
/usr/bin/mc mb myminio/test-bucket;  
/usr/bin/mc policy download myminio/test-bucket;  
exit 0;  
"
```

The **minio-setup** service in the Docker Compose snippet automates the initial setup for a MinIO server. Using the **minio/mc** image, it configures access to MinIO, deletes and recreates a bucket named **test-bucket**, and sets its policy to set the bucket as public.

Finally, at the end of the **docker-compose.yml** file, let us add the **minio\_data** volume:

```
volumes:  
  minio_data:
```

This configuration defines a Docker volume named **minio\_data**, which ensures that data stored by MinIO remains accessible across container restarts and deployments.

Let us restart **docker compose**:

```
docker compose restart
```

### INFO:

*Docker volumes are a mechanism for persisting data generated by and used by Docker containers. Unlike data stored in container layers, which is ephemeral and disappears when a container is removed, volumes are stored in a part of the host filesystem managed by Docker. Volumes can be easily created, managed, and mounted into containers using Docker commands or defined in Docker Compose files.*

We still need to update project settings in **src/recommendation\_system/settings.py** to set the AWS configuration to use the new **minio** local service:

```
AWS_ACCESS_KEY_ID = os.getenv("AWS_ACCESS_KEY_ID",  
"minioadmin")  
AWS_SECRET_ACCESS_KEY = os.getenv("AWS_SECRET_ACCESS_KEY",  
"minioadmin")  
AWS_STORAGE_BUCKET_NAME = os.getenv("AWS_STORAGE_BUCKET_NAME",  
"mybucket")
```

```
AWS_S3_ENDPOINT_URL = os.getenv("AWS_S3_ENDPOINT_URL",
"http://localhost:9000")
AWS_S3_CUSTOM_DOMAIN = os.getenv("AWS_S3_CUSTOM_DOMAIN",
"localhost:9000")
AWS_S3_USE_SSL = os.getenv("AWS_S3_USE_SSL", "True") == "True"
AWS_STORAGE_BUCKET_NAME = os.getenv("AWS_STORAGE_BUCKET_NAME",
"test-bucket")
```

The preceding code configures an S3 storage service using environment variables in Python. It retrieves credentials and connection details such as `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, the bucket name (`AWS_STORAGE_BUCKET_NAME`), and the endpoint URL (`AWS_S3_ENDPOINT_URL`) from the environment, providing default values for a local MinIO setup. The `AWS_S3_CUSTOM_DOMAIN` is set for accessing the storage with a specific domain, and `AWS_S3_USE_SSL` indicates whether to use SSL (HTTPS) for secure connections, defaulting to `True`. This setup is commonly used in applications that interact with object storage for file handling, allowing for development and production environments by simply changing environment variables.

Now, run the tests to verify that all of them pass:

```
DJANGO_SETTINGS_MODULE=recommendation_system.test_settings rye
run pytest -vvv -s

===== 27 passed in 1.68s
=====
```

Success! All tests are back to green.

Still, our file upload endpoints need a final refactor. If someone uploads a big file, one worker will consume the file, which could take several minutes. Celery tasks have time limits, but even if we increase these values, our task processing should take only a short time. In the next section, we will finally refactor to large datasets.

## [Handling Large Data Sets and Long-Running Processes](#)

In this section, we will refactor the upload endpoint to divide the problem into smaller parts, allowing us to scale and support big-size files.

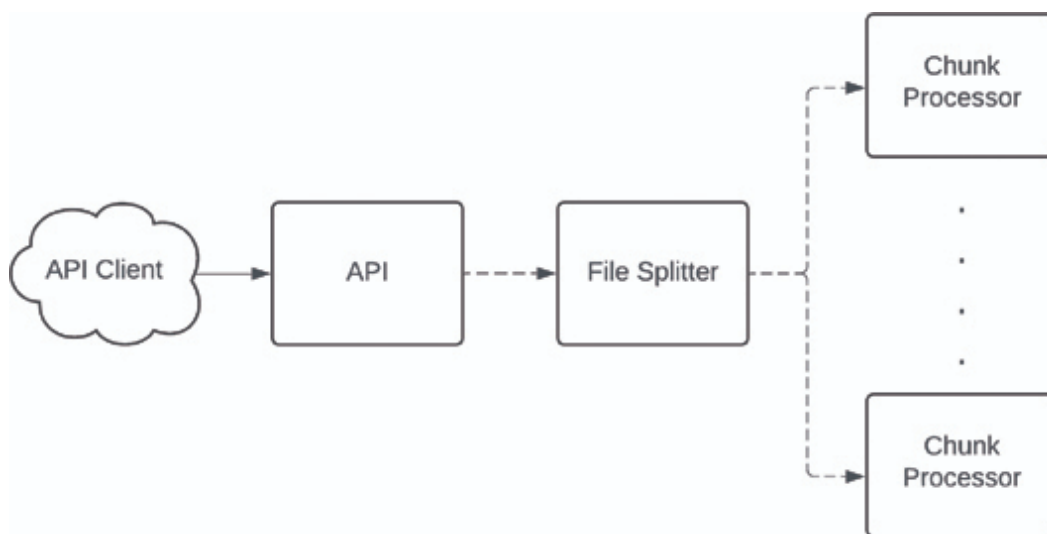
The refactor we want to do will open the big file and split it into chunks of smaller files. This will reduce the runtime of each task and allow more tasks to be executed simultaneously.

**INFO:**

*Parallelism and concurrency are strategies for handling multiple tasks. Parallelism executes tasks simultaneously, often on various processors, to speed up overall execution time by dividing work into smaller units. Concurrency involves managing and making progress on multiple tasks within the same timeframe, even if not executed simultaneously. Parallelism maximizes the performance through hardware, while concurrency addresses multitasking within software environments.*

The API will not change. However, the service layer entry point for processing files needs a refactor. The new process will first be to split the file into smaller chunks and enqueue these chunks to process them with a different worker. Let us introduce the new components to the architecture:

- **File Splitter Worker:** This worker will receive the whole file and split it into smaller chunks. Once the chunks are ready, it will enqueue all the workers for the Chunk worker
- **Chunk Worker:** There will be two types of chunk workers: the CSV and JSON. These workers will receive a URL containing the data to process.



**Figure 4.3:** New backend architecture

Celery has a feature that enables us to develop the architect described above. This feature is the workflow. Workflows allow developers to compose tasks in various patterns such as chains, groups, or chords. By supporting these patterns, Celery can handle everything from simple background tasks to complex data processing pipelines. Workflows enable the orchestration of task execution in several powerful patterns:

- **Chains** are sequences of tasks where each task starts after the previous one finishes, passing its result to the next in the chain.
- **Groups**: This pattern allows for the parallel execution of tasks. A group of tasks can be dispatched and executed across the available workers.
- **Chords**: A chord is a group of tasks with a callback. The tasks in the group are executed in parallel, and once all these tasks are completed, the callback is triggered with the results of the tasks.
- **Canvas**: The general term for combining these patterns to create complex workflows, allowing for the composition of tasks in ways that can handle dependencies, partial results, and synchronization.

The new architecture will be implemented using the chain and group to execute the new workflow.

The changes will be done in the service layer and Celery tasks. The **process\_file** task will orchestrate the splitting and parallel processing of the file chunks. The new task **split\_file\_task** will create the file chunks from the original file. The **split\_file\_task** will open the CSV or JSON file and generate smaller-size files that will be uploaded to the S3 service and returned as a list of paths. The **process\_file** with the list of paths will use the celery workflow chain to process the chunks. The **process\_chunks** task will receive the chunk filenames and use the group to process the chunks in parallel.

Let us have a look at the new **process\_file** task, open the **src/movies/tasks.py** and replace **process\_file** with the new implementation:

```
@shared_task
def process_file(file_name: str, file_type: str) -> int:
    """Orchestrates the splitting and parallel processing of file
    chunks."""
    if not default_storage.exists(file_name):
```

```

        raise ValidationError("File does not exist in storage.")

    # Chain split_file_task with the processing of chunks
    workflow = chain(
        split_file_task.s(file_name, file_type), # Splits the file
        process_chunks.s(file_type)             # Processes all
        chunks
    )
    result = workflow.apply_async()
    return result

```

This function takes two arguments: **file\_name** and **file\_type**, as before. The function first checks if the file exists in storage using **default\_storage.exists**. If the file does not exist, it raises a **ValidationError**.

#### INFO:

*The **s()** method is a shorthand for **signature()**, used to create an immutable signature for the task. This means the parameters you pass are bound to the task before execution.*

The function constructs a Celery workflow using a chain upon validating the file's existence. This workflow consists of two tasks: **split\_file\_task** and **process\_chunks**. The **split\_file\_task** is responsible for splitting the file into manageable chunks, passing the **file\_name** and **file\_type** as arguments. Following the file splitting, the **process\_chunks** task processes these chunks, taking **file\_type** as its argument to handle different processing methods based on the file type.

Once **split\_file\_task** has been completed and the paths to the file chunks are returned, these paths are automatically passed as the first argument to **process\_chunks**.

The workflow is then executed asynchronously with **apply\_async()**. The function returns a Celery **AsyncResult** object, which can be used to monitor or retrieve the outcome of the task chain.

Here is the implementation of **process\_chunks**:

```

@shared_task
def process_chunks(chunk_paths: list, file_type: str) -> int:

```



```

"""Task to handle processing of each chunk. This is called
after the file has been split."""
# Create a group of tasks to process each chunk
task_group = group(process_chunk.s(chunk_path, file_type) for
chunk_path in chunk_paths)
return task_group.apply_async() # Apply group asynchronously
and return the AsyncResult of the group

```

The Celery task **process\_chunks** uses Celery's **group** function to manage multiple tasks concurrently, where each task processes a separate file chunk. This setup distributes the workload across multiple workers. Each task is prepared with specific chunk paths and a file type, ensuring that processing is optimized for the content type. The function returns an **AsyncResult** object.

Then **process\_chunk** is similar to the previous implementation of **process\_file**:

```

@shared_task
def process_chunk(chunk_path: str, file_type: str) -> int:
    """Processes a single file chunk."""
    with default_storage.open(chunk_path, "r") as file:
        if file_type == "text/csv":
            result = parse_csv(file)
        elif file_type == "application/json":
            result = parse_json(file)
        else:
            raise ValidationError("Invalid file type")
    return result

```

**process\_chunk** processes individual file chunks based on their file type. It starts by opening the file using Django's **default\_storage**. The function then checks the file type: if it is a CSV, it calls **parse\_csv(file)**; if it is a JSON, it calls **parse\_json(file)**. If the file type is neither, it raises a **ValidationError**.

Finally, we have the code of our file splitter:

```

@shared_task
def split_file_task(file_name: str, file_type: str) ->
list[str]:
    if file_type == "text/csv":

```

```

    result = split_csv_file(file_name)
elif file_type == "application/json":
    result = split_json_file(file_name)
else:
    raise ValidationError("Invalid file type")

return result

```

The Python code defines a Celery task named **split\_file\_task** that splits files based on their type, either CSV or JSON. If an unrecognized file type is provided, it raises a **ValidationError**. The implementation is very similar to **process\_chunk**.

Let's have a look to the **split\_csv\_file** function:

```

def split_csv_file(file_path: str, chunk_size_mb: int = 100) ->
list[str]:
    chunk_paths = []
    part = 1
    current_chunk_size = 0
    chunk_lines = []

    with default_storage.open(file_path, "r") as file:
        for line in file:
            line_size = len(line.encode("utf-8"))
            if current_chunk_size + line_size > chunk_size_mb * 1024 *
1024:
                chunk_file_name = f"{file_path}_part_{part}.csv"
                default_storage.save(chunk_file_name,
ContentFile("".join(chunk_lines)))
                chunk_paths.append(chunk_file_name)
                chunk_lines = [line]
                current_chunk_size = line_size
                part += 1
            else:
                chunk_lines.append(line)
                current_chunk_size += line_size

    if chunk_lines: # Save the last chunk if there is any
        chunk_file_name = f"{file_path}_part_{part}.csv"

```

```

default_storage.save(chunk_file_name,
ContentFile("".join(chunk_lines)))
chunk_paths.append(chunk_file_name)

return chunk_paths

```

The **split\_csv\_file** function handles CSV files by slicing them into small chunks. It reads the file line by line, bundling the lines together until it hits the chunk size, and then writes these bundles out as individual smaller files. The function creates each new file segment with a distinct name and processes the whole file until it's fully divided. The last if condition saves the previous chunk, ensuring the entire file is split.

The **split\_json\_file** follows a similar approach:

```

def split_json_file(file_path: str, chunk_size_mb: int = 100) -
> list[str]:
    chunk_paths = []
    part = 1
    current_chunk_size = 0
    chunk_objects = []

    with default_storage.open(file_path, "r") as file:
        objects = json.load(file)

    for obj in objects:
        obj_str = json.dumps(obj)
        obj_size = len(obj_str.encode("utf-8"))
        if current_chunk_size + obj_size > chunk_size_mb * 1024 *
1024:
            chunk_file_name = f"{file_path}_part_{part}.json"
            default_storage.save(
                chunk_file_name, ContentFile(json.dumps(chunk_objects))
            )
            chunk_paths.append(chunk_file_name)
            chunk_objects = [obj]
            current_chunk_size = obj_size
            part += 1
        else:
            chunk_objects.append(obj)
            current_chunk_size += obj_size

```

```

        if chunk_objects: # Save the last chunk if there is any
            chunk_file_name = f"{file_path}_part_{part}.json"
            default_storage.save(chunk_file_name,
                                ContentFile(json.dumps(chunk_objects)))
            chunk_paths.append(chunk_file_name)

    return chunk_paths

```

The **split\_json\_file** function breaks down large JSON files into smaller ones. It reads the entire JSON file into memory and then iterates through each object. For each object, the function checks if adding it to the current chunk would exceed the specified size limit (in megabytes). If so, it saves the current chunk as a new file, resets the tracking variables, and fills the next chunk. This process repeats until every object has been allocated to a chunk. Splitting a large JSON file could be challenging since our implementation will read the entire JSON file into memory, which was not required for CSV.

Now, run the tests to verify that all of them pass:

```

DJANGO_SETTINGS_MODULE=recommendation_system.test_settings rye
run pytest -vvv -s
===== 27 passed in 1.68s
=====

```

Success! All tests are back to green.

It is important to note that after all our refactoring, our test suite should be improved to add more cases. For example, the test **test\_general\_upload\_view** uses tiny files to verify the file upload; however, our implementation uses chunks, and the tests do not cover edge cases.

If you want to test it with curl, try the following command:

```

curl -X POST -F
"file=@/Users/mandarina/workspace/django_rest_book/src/all_movies_data.csv;type=text/csv" -H "Content-Type: multipart/form-data" http://localhost:8000/api/v1/movies/upload/

```

The execution time of the file uploaded using the **all\_movies\_data.csv** example file should take seconds instead of several minutes since the file will be processed in the background.

If you try with bigger files, you will quickly find scalability issues with the Django's default database configuration:

```
django.db.utils.OperationalError: database is locked
```

This is something we will improve in the next chapter.

## Conclusion

In this chapter, we iterated a naive file upload implementation into a more scalable solution by performing iterative refactors. After each refactor, our tests were executed until the test was passed. By making incremental improvements, we could tackle the complexity of the current implementation. We could scale our upload endpoint; however, we quickly hit the wall with the default Django database, SQLite, which does not allow concurrent writes to the database and will not work in a distributed environment.

In the next chapter, we will add security to our API and prepare the server for scalability by changing the database.

## Questions

1. What is the primary disadvantage of using a synchronous backend for large datasets or complex tasks?
2. What are the potential issues with using a local filesystem for file processing in distributed systems?
3. How does Celery ensure that a task is only processed once, even if submitted multiple times?
4. What is the difference between Celery's task chaining and task grouping, and how do they contribute to complex workflows?

## Exercises

1. Write unit tests for the `split_file_task` to ensure it correctly splits files into specific sizes. Include tests for edge cases such as empty files, files smaller than the chunk size, and files strictly at the chunk limit.
2. Create a functional test that simulates the complete process of uploading a file, splitting it, and processing the chunks. This should

verify that the chunks are correctly processed and the final output meets expected conditions.

3. Create a workflow using Celery that involves chaining two tasks. The first task processes a file and outputs some data; the second task stores this data in a database.

[OceanofPDF.com](http://OceanofPDF.com)

# CHAPTER 5

## Securing and Scaling Data Science APIs

### Introduction

In modern web development, securing and scaling data science APIs is a critical aspect of building robust applications. This chapter explores essential concepts and practices focusing on authentication, middleware, and security measures necessary for maintaining reliable and protected data APIs.

### Structure

This chapter covers the following topics:

- Understanding Django's Authentication System
- Introduction to Django's Middleware
- Setting Up Django Admin
- Groups and Permissions
- Implementing Authentication Methods
- Custom Permissions for Sensitive Data
- Ensuring Security in Data APIs

### Understanding Django's Authentication System

There are two essential concepts in Django's authentication system: authentication and authorization.

Authentication occurs when a user logs in using username and password credentials. Authorization, on the other hand, involves the server verifying permissions to allow the requested operation. For instance, an Editor attempting to delete a Task without proper permissions will encounter an authorization error.

**INFO:**

*Authentication is the process of verifying the identity of a user or entity. Authentication answers the question, “who are you?”.*

*Authorization occurs after authentication and determines what an authenticated user is allowed to do. Authorization answers, “Are you allowed to access or do?”*

*There are specific HTTP response status codes, 401 and 403, respectively for denied authentication and authorization.*

The Django REST Framework provides HTTP authentication and authorization features, which we will explore in this chapter.

The HTTP protocol is stateless, treating each server request as an independent event. It requires a mechanism to maintain authentication across multiple requests. In many web applications, servers create cookies to store client-side information, allowing the identification of authenticated users via session identifiers. Unlike session authentication, REST APIs typically use a different approach due to machine-to-machine communication. The most common methods include:

- **Basic Authentication:** This authentication method encodes the username and password using base64. It is easy to implement but only the most secure if combined with SSL/TLS since credentials are sent in every request.
- **Token Authentication:** In this method, credentials are exchanged for a token during the login process. This token is sent in the header on every request. The token can expire and be stored on the server. This method is more secure than basic authentication and is widely used due to its simplicity.
- **JWT (JSON Web Token):** JWT includes more information in the token, such as user data and expiration time. Thanks to this additional information in the token, the server can verify its validity without needing to query a database.
- **OAuth:** This method supports delegated authorization. It is widely used for scenarios where an application needs to perform actions on behalf of the user without accessing the user’s password.



## Introduction to Django's Middleware

Before diving deep into the authentication system, it is essential to understand middleware first.

Think of middleware as layers inserted before processing each view or after a view has responded.

These middlewares are configured under the **MIDDLEWARE** section in the **recommendation\_system/settings.py** file. **MIDDLEWARE** configuration is a list, and the framework will execute each middleware in the order of this list for requests but then in the reverse order for the response of processing.

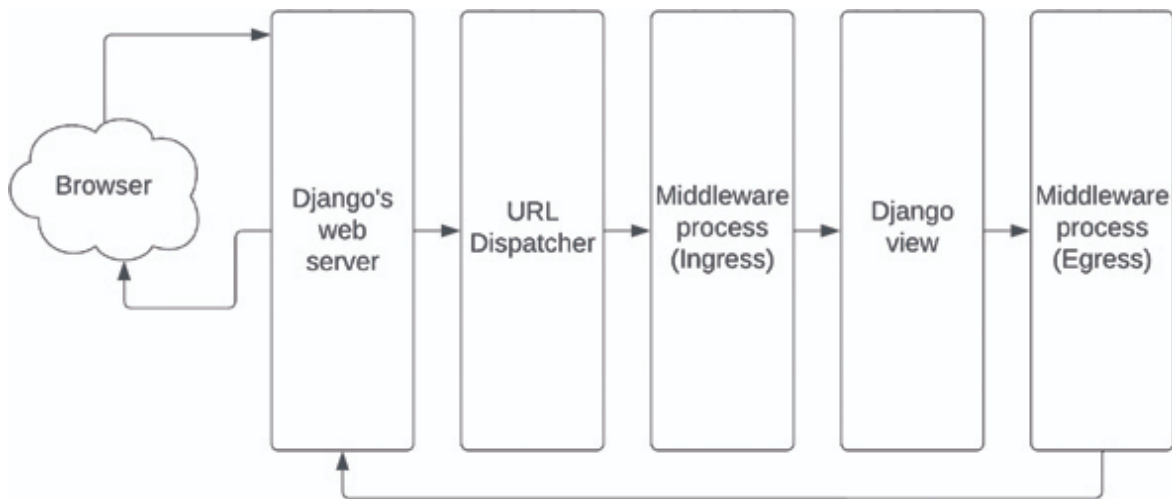
Middleware can alter requests or responses, enhance security, manage redirects, or include data for views. It can also handle exceptions when conditions are not met, making it ideal for authentication and authorization checks.

Now, let us suppose the scenario where the user navigates to the Movie view and makes an HTTP request to the server.

The server received the request and will go through a chain of middleware. The default **MIDDLEWARE** configuration looks like the following:

```
MIDDLEWARE = [  
    "django.middleware.security.SecurityMiddleware",  
    "django.contrib.sessions.middleware.SessionMiddleware",  
    "django.middleware.common.CommonMiddleware",  
    "django.middleware.csrf.CsrfViewMiddleware",  
    "django.contrib.auth.middleware.AuthenticationMiddleware",  
    "django.contrib.messages.middleware.MessageMiddleware",  
    "django.middleware.clickjacking.XFrameOptionsMiddleware",  
]
```

Each layer evaluates the request as it passes through the middleware chain. If a layer detects a problem, it can stop further processing and return the appropriate status code.



*Figure 5.1: HTTP Request through Django's Architecture*

The **SecurityMiddleware** checks for generic security configurations, like secure SSL redirects. By the end of the chapter, we will review the best security practices deeply.

Once the **SecurityMiddleware** checks are successful, the framework will continue with the next middleware in the list, the **SessionMiddleware**. The **SessionMiddleware** provides a way to store data in the session and makes this data available in the request object.

**CommonMiddleware** handles redirects based on URL trailing slashes. Additionally, it uses the **PREPEND\_WWW** setting and appends a content-length header.

Django protects against Cross-site request forgery using the middleware **CsrfViewMiddleware**. This middleware checks if the requests are safe based on the token verification. This middleware uses the token we set in forms, covered in the previous chapter when we added `{% csrf_token %}`.

When **AuthenticationMiddleware** is enabled in the settings, it associates the corresponding user, if any, with the request object. If the user is authenticated, the session store will locate the user. If the user is not authenticated, it will assign an instance of **AnonymousUser**.

The Django framework allows users to see one-time informational messages. The middleware **MessageMiddleware** stores these messages in the request.

Lastly, **XFrameOptionsMiddleware** protects against clickjacking by setting the X-Frame-Options header, dictating if browsers can render a page within tags such as `<frame>`, `<iframe>`, `<embed>`, or `<object>`.

**INFO:**

*Clickjacking is a malicious technique where an attacker tricks a user into clicking something different than the user perceives. Clickjacking can lead to unintended actions on a different application without the user's consent.*

*When using the X-Frame-Options, the browser will allow or not to render a page inside an `<iframe>`.*

## [Understanding Django Middleware](#)

The framework makes it very easy to create your custom middleware. You will need to create a new class and implement some methods.

We will create middleware in Django to measure request render times, which will help us address any efficiency issues.

Create a new file **tasks/middlewares.py** with the following code:

```
import time
import logging
logger = logging.getLogger(__name__)

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

    def __call__(self, request):
        # Start the timer when a request is received
        start_time = time.time()

        # Process the request and get the response
        response = self.get_response(request)

        # Calculate the time taken to process the request
        duration = time.time() - start_time

        # Log the time taken
        logger.info(f"Request to {request.path} took {duration:.2f} seconds.")

        return response
```

The class defines `__init__` and `__call__` methods, where `__init__` receives the `get_response` function to obtain the response later.

In Python, the `__call__` method allows class instances to be callable, meaning they can be invoked using parentheses, like functions.

The `__call__` method records the current time, gets the response, and then calculates the duration it took to process `get_response`. For logging, we use an f-string, a formatted string literal. The brackets allow the value of a variable to be inserted into the string. This syntax also allows the format's specification using a colon and some configuration. In the preceding example, we use `.2f` to format the value as a floating-point number with exactly two decimal places.

Finally, the code returns the response for the next middleware or view.

You can now add the `RequestTimeMiddleware` to the `MIDDLEWARE` setting in the `settings.py` file:

```
MIDDLEWARE = [  
    "movies.middlewares.RequestTimeMiddleware",  
    ...  
]
```

Your custom middleware is ready to use! Note that the new middleware is placed at the beginning of the list. It will be the first to be called on an incoming request and the last on the given response object.

If you want to manipulate the response, you can easily add data after calling the `get_response`. If you want to add a new header, here is an example:

```
def __call__(self, request):  
    response = self.get_response(request)  
    response["X-Custom-Header"] = "This is a custom header value"  
    return response
```

## [Setting Up the Django Admin](#)

Once our project has authentication and authorization enabled, we need a way to create users and add permission to the users. Django admin is an administrative interface provided by the framework.

To use Django admin, you need to perform some basic setup:

Open the file `src/recommendation_system/settings.py` and ensure that the Django admin is added to the **INSTALLED\_APPS**:

```
INSTALLED_APPS = [  
    ...  
    "django.contrib.admin",  
    ...  
]
```

Now, we need to create the first admin user. You can create one by running the following management command in the terminal:

```
rye run python src/manage.py createsuperuser
```

The command will prompt you to create a username, email, and password.

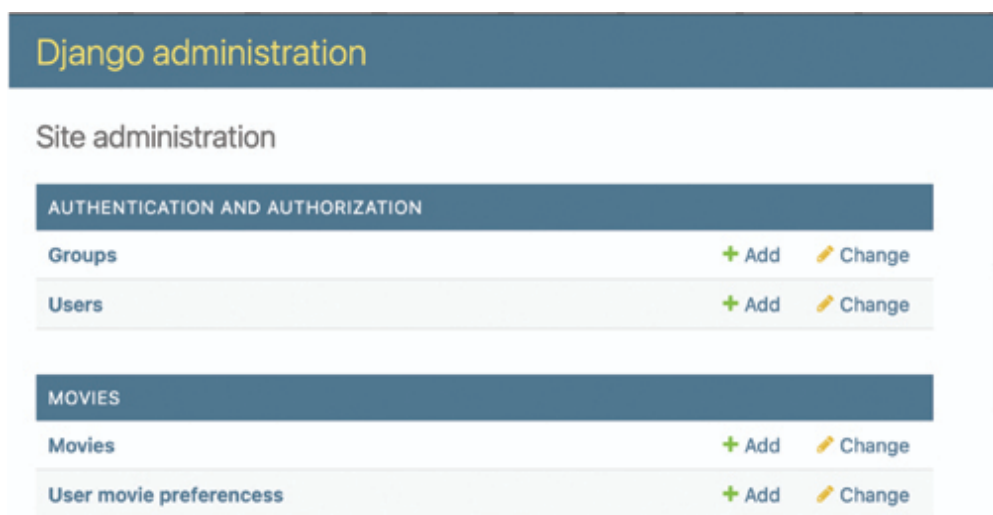
Now we need to register the models, open the file `src/movies/admin.py` and register the **Movie** and **UserPreferences** model:

```
from django.contrib import admin  
from .models import Movie, UserMoviePreferences  
  
admin.site.register(Movie)  
admin.site.register(UserMoviePreferences)
```

Now, start the server:

```
rye run python src/manage.py runserver
```

Open the URL `http://localhost:8000/admin` in the browser, and log in with the superuser created before. You should see the following admin menu:



*Figure 5.2: Django admin main menu*

## Groups and Permissions

In Django, groups and permissions are fundamental concepts for managing access control within web applications. Permissions define what actions users can perform on different parts of the application, while groups are collections of permissions that can be assigned to users to manage their access rights efficiently.

Here is a list of common Django permission classes:

- **IsAuthenticated:** This permission class ensures that the user requesting access is authenticated, meaning they have provided valid credentials and are logged into the system.
- **IsAdminUser:** Users with this permission have full administrative access. They can perform any action within the system, including managing other users, groups, and permissions.
- **AllowAny:** This permission class allows unrestricted access to the associated view or API endpoint. It is commonly used for public or anonymous access scenarios where authentication is not required.
- **IsAuthenticatedOrReadOnly:** With this permission class, authenticated users have full access to perform actions, while unauthenticated users are only allowed read-only access. This is useful for APIs where some resources should be publicly accessible, but modifications require authentication.
- **Django Model Permissions** (for example, **IsOwner**): These permissions are custom-defined based on the application's specific requirements. For example, **IsOwner** might restrict actions to the owner of a particular resource, such as a user being allowed to modify only their profile.

These permissions can be assigned at both the view level and the model level in Django, providing flexibility in controlling access across different parts of the application. Django also supports custom permissions, allowing developers to define and enforce access rules tailored to their application's needs. Properly configuring groups and permissions ensures that applications are secure and operate according to the intended access control policies.

## Implementing Authentication Methods

Django Rest Framework (DRF) supports multiple authentication methods out of the box. We will cover some of the most commonly used methods: Basic Authentication, Token Authentication, and JWT (JSON Web Token) Authentication. Each method serves different use cases, and choosing the right one depends on your application's requirements.

### Basic Authentication

Basic Authentication is the simplest form of authentication. It involves sending the username and password encoded in Base64 with each HTTP request. While easy to implement, it is not very secure unless used over HTTPS, as credentials are exposed in every request.

Let us enable **BasicAuthentication** in DRF, and update your **src/recommendation\_system/settings.py** to include authentication classes:

```
REST_FRAMEWORK = {  
    ...  
    'DEFAULT_AUTHENTICATION_CLASSES':  
        ['rest_framework.authentication.BasicAuthentication', ],  
}
```

Now open your api views located in the file **src/movies/api.py** and add the **permission\_classes** set to **[IsAuthenticated]**

```
from rest_framework.permissions import IsAuthenticated  
class MovieListCreateAPIView(generics.ListCreateAPIView):  
    permission_classes = [IsAuthenticated]  
    ...
```

**IsAuthenticated:** Ensures that only authenticated users can access the view.

You have to add this to all the API views you want to force authentication.

Now, you will need to provide a username and password. DRF will use your Django users. Most API clients (like Postman) can handle Basic Authentication.

If you open your browser, the api to list movies will ask you for credentials:



*Figure 5.3: Credentials not provided error*

### INFO:

*You can create an API user by logging into the admin `localhost:8000/admin/auth/user/add`*

## Token Authentication

Token Authentication is a more secure and flexible way to handle authentication. The server issues a token after a successful login, which is then used for subsequent requests.

Update your settings file located at `src/recommendation_system/settings.py`:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'rest_framework.authtoken',
    ...
]
```

Since we have added a new application, we need to run the migrations:

```
cd src
rye run python manage.py migrate
```

In the same file, update rest framework settings to use token authentication:

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
}
```



Now we need to add an API endpoint to get the authentication token. Before adding, let us discuss where it should be located. Right now, the Django app `movies` have an API, but the token API is not relevant. We need to create a new Django application where we are going to add the new authentication endpoints. Let us create the new application by executing:

```
cd src
rye run python manage.py startapp api_auth
```

Add the new application to your `src/recommendation_system/settings.py`:

```
INSTALLED_APPS = [
    ...
    'api_auth',
    ...
]
```

Create a new file located at `src/auth/urls.py` file and add the new endpoint to obtain the token:

```
from django.urls import path
from rest_framework.authtoken.views import obtain_auth_token

urlpatterns = [
    path('token/', obtain_auth_token, name='api_token_auth'),
]
```

Include the Auth URLs in the main project URLs. Open your main project `urls.py` and include the auth application URLs:

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

urlpatterns = [
    path("admin/", admin.site.urls),
    path("auth/", include("auth.urls")),
    path("api/v1/movies/", include("movies.urls")),
]
```

Let us test the obtain token api by using **curl**:

```
$ curl -X POST -d "username=mandarina&password=123"
http://localhost:8000/auth/token/ | jq .
{"token": "8475024c8a0f30db66612c0ee70f45d0da294b4d"}
```

With the new token, we can list the movies:

```
curl -H "Authorization: Token
8475024c8a0f30db66612c0ee70f45d0da294b4d"
http://localhost:8000/api/v1/movies/ | jq .
```

**INFO:**

*jq is a command-line tool for processing and manipulating JSON data. It allows you to extract, filter, and format JSON content efficiently.*

Your API has now token-based authentication!

## JWT (JSON Web Token) Authentication

JWT (JSON Web Token) is a compact, URL-safe means of securely transmitting information between parties as a JSON object. It consists of three parts: a header, a payload, and a signature, each encoded as Base64. JWTs are commonly used for authentication and information exchange in web applications. They are cryptographically signed to ensure integrity, enabling servers to verify the authenticity of tokens without storing the session state. This self-contained nature makes JWTs scalable and portable across different systems and platforms, while also mitigating the need for frequent database queries during authentication.

First, install SimpleJWT for DRF:

```
rye add django-rest-framework-simplejwt
```

Open `src/recommendation_system/settings.py` and add `'rest_framework_simplejwt':`

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'rest_framework_simplejwt',
    ...
]
```

In the same file, set the authentication classes:

```
REST_FRAMEWORK = {
    ...
```

```

'DEFAULT_AUTHENTICATION_CLASSES': [
    'rest_framework_simplejwt.authentication.JWTAuthentication',
],
...
}

```

SimpleJWT does not require any migrations.

Now, we need to add two endpoints for obtaining the token and the refresh token.

Open the URLs file of the **api\_auth** application, **src/api\_auth/urls.py**:

```

from django.urls import path
from rest_framework_simplejwt.views import (
    TokenObtainPairView, TokenRefreshView, )
from rest_framework.authtoken.views import obtain_auth_token

urlpatterns = [
    path('token/', obtain_auth_token, name='api_token_auth'),
    path('jwt_token/', TokenObtainPairView.as_view(),
        name='token_obtain_pair'),
    path('jwt_token/refresh/', TokenRefreshView.as_view(),
        name='token_refresh'),
]

```

Here is an example of how to obtain the JWT using **curl**:

```

curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"username": "your_username", "password": "your_password"}' \
  http://localhost:8000/auth/api/token/

```

Here is the JSON response:

```

{
  "refresh": "eyJhbGciOiJIUz...",
  "access": "eyJhbGciOiJIUz..."
}

```

Here is an example **curl** command to obtain the movies using the JWT:

```

curl -X GET \
  -H "Authorization: Bearer "eyJhbGciOiJIUzI1NiIsIn..."
  http://localhost:8000/api/movies/

```

To obtain the refresh token, it is required to provide the fresh token returned by the JWT token api:

```
curl -X POST \
  -H "Content-Type: application/json" \
  -d '{"refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."}' \
  http://localhost:8000/auth/jwt_token/refresh/
```

The API will return a new access token:

```
{"access": "eyJhbGciOiJIUzI1NiIsInR5cCI6I..."}
```

Implementing JWT authentication in Django involves installing **django-rest-framework-simplejwt**, configuring the authentication settings in settings.py, defining endpoints for token management, and integrating these endpoints into your project's URL configuration. This setup provides a stateless and scalable way to authenticate users in your Django application.

JWT tokens offer several advantages for modern web applications. Their stateless nature simplifies scaling and eliminates the need for server-side storage. JWTs support interoperability across different platforms and domains. Security-wise, they can be signed to ensure data integrity and prevent tampering. JWTs also provide flexibility in payload content, allowing for the inclusion of user information and permissions. Overall, JWT tokens streamline authentication, enhance security, and optimize performance.

## Custom Permissions for Sensitive Data

In Django, custom permissions can be configured to restrict access to sensitive data and operations within your application. Permissions are typically assigned to users based on roles or groups defined in the Django admin interface. This section explores how to define and manage custom permissions for managing movies API using Django's built-in capabilities.

### **Creating Groups and Permissions:**

Django admin allows administrators to define custom groups and assign permissions to these groups.

Navigate to the Django admin interface (<http://localhost:8000/admin>) and log in with your superuser credentials.

Under the **Auth** section, select **Groups**.

Create a new group, for example, **Movie Editors**, and assign permissions such as "Can view movie", "Can add movie", "Can change movie", and "Can delete movie".

Assigning Permissions to a group:

The screenshot shows the 'Add group' form in the Django admin interface. The 'Name' field is filled with 'Movie Editors'. Under the 'Permissions' section, there are two panels: 'Available permissions' and 'Chosen permissions'. The 'Available permissions' panel has a search bar with 'Movie' entered and a list of permissions: 'Movies | user movie preferences | Can add user movie preferen', 'Movies | user movie preferences | Can change user movie prefe', 'Movies | user movie preferences | Can delete user movie prefer', and 'Movies | user movie preferences | Can view user movie prefer'. The 'Chosen permissions' panel has a search bar with 'Filter' and a list of permissions: 'Movies | movie | Can add movie', 'Movies | movie | Can change movie', 'Movies | movie | Can delete movie', and 'Movies | movie | Can view movie'. At the bottom, there are 'Choose all' and 'Remove all' buttons. A note at the bottom states: 'Hold down "Control", or "Command" on a Mac, to select more than one.'

*Figure 5.4: Admin interface to configure group permissions*

Once groups are defined, users can be assigned to these groups.

Navigate to "Users" in the admin interface and select the user you wish to assign permissions to.

In the user detail view, select the appropriate group ("Movie Editors") from the available groups.

The screenshot shows the user detail view in the Django admin interface. The URL bar shows 'localhost:8000/admin/auth/user/1/change/'. The left sidebar has a list of users with 'Add' buttons. The main content area has a 'Groups' section. It includes checkboxes for 'Staff status' and 'Superuser status', both of which are checked. Below these, there are two panels: 'Available groups' and 'Chosen groups'. The 'Available groups' panel has a search bar with 'Filter' and is empty. The 'Chosen groups' panel has a search bar with 'Filter' and contains the group 'Movie Editors'. At the bottom, there are 'Choose all' and 'Remove all' buttons. A note at the bottom states: 'The groups this user belongs to. A user will get all permissions granted to each of their groups. Hold down "Control", or "Command" on a Mac, to select more than one.'

*Figure 5.5: Admin interface to add groups to a user*

We still need to integrate Django permissions with the API views. In your Django application's movie API views (`src/movies/api.py`), integrate these permissions by associating them with specific views using the Django REST Framework.

The first step is to create a custom model permission class to map the Django permissions to the API verbs GET, POST, PUT, PATCH, and DELETE. Create a new file located at `src/api_auth/permissions.py` with the **CustomDjangoModelPermissions** new class:

```
from rest_framework.permissions import DjangoModelPermissions

class CustomDjangoModelPermissions(DjangoModelPermissions):
    """
    Custom Django model permissions class that maps Django model
    permissions to HTTP methods.
    """
    perms_map = {
        'GET': ['%(app_label)s.view_%(model_name)s'],
        'OPTIONS': [],
        'HEAD': [],
        'POST': ['%(app_label)s.add_%(model_name)s'],
        'PUT': ['%(app_label)s.change_%(model_name)s'],
        'PATCH': ['%(app_label)s.change_%(model_name)s'],
        'DELETE': ['%(app_label)s.delete_%(model_name)s'],
    }
```

To understand how this new class works, it is important to understand how Django permissions work.

Django permissions are a way to control access to different parts of a web application built using Django. They help determine what actions users are allowed to perform on specific models within an application.

Let us break down the example permission string `'%(app_label)s.view_%(model_name)s'`:

- **%(app\_label)s**: This part refers to the application label, which is a unique identifier for each Django application within a project. It is usually the name of the directory containing the application's code. For

example, if you have an application named **'movies'** in your Django project, its app label would typically be **'movies'**.

- **view\_**: This prefix indicates the specific action or permission being defined. In this case, **'view\_'** suggests that this permission grants the ability to view (read) instances of a particular model.
- **%(model\_name)s**: Here, **'model\_name'** refers to the name of a Django model within the specified application (**app\_label**). Models in Django represent database tables and define the structure of data storage. For instance, if your application has a model named **'Movie'**, **'model\_name'** would be replaced with **'Movie'**.

In our movies Django application, we have the app label **movies** and a model named **Movie**, the permission **'%(app\_label)s.view\_%(model\_name)s'** would translate to **'movies.Movie.view\_post'**. This permission would allow users who have it to view instances of the **Movie** model within the **movies** application.

```
from rest_framework.permissions import IsAuthenticated
from rest_framework import generics
from .models import Movie
from .serializers import MovieSerializer
from api_auth.permissions import CustomDjangoModelPermissions

class MovieListCreateAPIView(generics.ListCreateAPIView):
    queryset = Movie.objects.all()
    serializer_class = MovieSerializer
    permission_classes = [IsAuthenticated,
                          CustomDjangoModelPermissions]
```

### INFO:

*In Django REST Framework (DRF), generics are pre-built components that simplify API development by handling common tasks like CRUD operations, pagination, and serialization. They include base classes like **APIView** and **GenericAPIView**, along with mixins such as **ListAPIView** and **RetrieveAPIView**. Generics reduce boilerplate code, enforce RESTful practices, and ensure consistency across API endpoints, enabling developers to focus more on business logic and less on repetitive tasks.*

Now, if we try to use our movies list API, we will get a 403 forbidden error:

```
curl -X GET \
  -H "Authorization: Token
  eb0bd1d8e3db5f4bbf541a8501613306ee46c61a"
  http://localhost:8000/api/v1/movies/
{"detail":"You do not have permission to perform this
action."}%
```

Adding the user to the group will allow it to list movies.

Custom permissions in Django provide a powerful way to control access to sensitive data and operations within your API. By defining groups and assigning permissions via Django admin, and integrating these permissions with Django REST Framework views, you can enforce fine-grained access control for your movie management API. This ensures that only authorized users can perform specific actions based on their roles or groups within the application.

In Django Rest Framework (DRF), permission classes are a critical component for securing API endpoints by defining who has access to specific resources and actions. For more granular control, **DjangoModelPermissions** integrates with Django's model-level permissions to enforce permissions like add, change, and delete based on user roles. Moreover, developers can create custom permission classes by subclassing **BasePermission** and implementing custom logic in the **has\_permission** and **has\_object\_permission** methods, allowing for highly tailored access control that can incorporate complex business rules and user attributes. By leveraging these permission classes, DRF ensures that APIs are robustly secured, providing appropriate access to users based on their authentication and authorization status.

We still have some API views that are unauthenticated, to add authentication for these views we will use a decorator **permission\_classes**.

In Django Rest Framework (DRF), the **@permission\_classes** decorator is used to specify the permissions that apply to a particular view or viewset. This decorator allows you to assign one or more permission classes directly to a view, ensuring that only users who meet the specified criteria can access the view. The decorator is placed above the view class definition and takes a list of permission classes as its argument.

When a request is made to a view, DRF checks the request against each permission class in the order they are listed. If any permission class denies



access, the request is denied, and an appropriate HTTP response is returned (typically 403 Forbidden or 401 Unauthorized). This allows for fine-grained control over access to different parts of an API, making it easy to enforce security policies.

Open the file `src/movies/api.py` and add the decorator to the classes **UserPreferencesView** and **WatchHistoryView**:

```
from rest_framework.decorators import permission_classes

# View to add new user preferences and retrieve them
@permission_classes([IsAuthenticated])
class UserPreferencesView(APIView):
    ...

# View to retrieve and add movies to the user's watch history
@permission_classes([IsAuthenticated])
class WatchHistoryView(APIView):
    ...
```

We still need to add permission to the **GeneralUploadView**, for this view in particular we only want to give access to admins. We can use the decorator with the permission **IsAdminUser**:

```
from rest_framework.permissions import IsAuthenticated,
IsAdminUser, DjangoModelPermissions

# View for general file uploads, restricted to admin users only
@permission_classes([IsAdminUser])
class GeneralUploadView(APIView):
    ...
```

We have all our views with authentication and permissions checks!

## [Ensuring Security in Data APIs](#)

Securing data APIs is critical to protecting sensitive information and ensuring that only authorized users can access specific resources. In this section, we will cover best practices and essential strategies for securing data APIs in Django, focusing on various aspects such as authentication, authorization, data validation, and secure communication.

In this section, we will review some good practices to maintain your API secure.

## Enforce HTTPS

Ensure that your API endpoints are only accessible over HTTPS. This encrypts the data exchanged between the client and server, protecting against man-in-the-middle attacks. Django provides several settings to enforce HTTPS.

Open your project settings located at `src/recommendation_system/settings.py` and add the new settings:

```
SECURE_SSL_REDIRECT = True
SECURE_HSTS_SECONDS = 3600
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True
```

These settings ensure that all HTTP traffic is redirected to HTTPS and that browsers enforce HTTPS for the specified period.

## Use Secure Authentication Methods

Always prefer secure authentication methods. For example, use Token Authentication or JWT over Basic Authentication, and ensure tokens are transmitted securely:

- **Token Authentication:** Use `TokenAuthentication` for simplicity and moderate security.
- **JWT Authentication:** Use `JWTAuthentication` for stateless, scalable authentication.

## Implement Rate Limiting

Rate limiting helps protect your API from abuse by limiting the number of requests a client can make in a given time. The `django-ratelimit` package can be used to implement rate limiting in Django:

```
from django_ratelimit.decorators import ratelimit

@ratelimit(key='ip', rate='5/m', method='GET', block=True)
def my_view(request):
    # Your view logic here
    pass
```

## Validate Input Data

Ensure all input data is validated to prevent injection attacks, such as SQL injection or script injection. Django's forms and serializers provide robust validation mechanisms. The use of Django serializers helps to reduce potential attacks.

## Implement Proper CORS Handling

Cross-Origin Resource Sharing (CORS) should be configured to allow only trusted domains to access your API. Use the **django-cors-headers** package to set this up.

First install the **django-cors-headers**:

```
rye add django-cors-headers
```

Open your project settings and update to enable CORS:

```
# settings.py
INSTALLED_APPS = [
    ...
    'corsheaders',
    ...
]
MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    ...
]
CORS_ALLOWED_ORIGINS = [
    "https://example.com",
    "https://sub.example.com",
]
```

## Limit Data Exposure

Always limit the amount of data exposed via your API. Use serializers to control which fields are returned and ensure sensitive data is never exposed:

```
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
```

```
fields = ['username', 'email'] # Do not expose sensitive
fields
```

## **Log and Monitor API Usage**

Implement logging and monitoring to keep track of API usage and detect any unusual activities. Django provides a logging framework that can be configured to log API requests and errors:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',

'class': 'logging.FileHandler',

            'filename': 'debug.log',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

## **Use Strong API Keys and Secrets Management**

Ensure that API keys and secrets are stored securely and are not hard-coded in your source code. Use environment variables or dedicated secret management services:

```
import os
API_KEY = os.getenv('API_KEY')
```

## **Regularly Update Dependencies**

Keep your Django application and its dependencies up to date to ensure you have the latest security patches. Use tools like pip-tools to manage dependencies in GitHub for automatic updates.

## **Conduct Security Audits and Penetration Testing**

Regularly audit your API for security vulnerabilities and conduct penetration testing to identify potential weaknesses. This helps in proactively addressing security issues before they can be exploited.

## **Conclusion**

This chapter covers securing and scaling data science APIs with Django and Django REST Framework. It explores authentication methods like Basic Authentication, Token Authentication, and JWT Authentication, emphasizing their implementation and security best practices. Middleware's role in request processing and security enhancements is also discussed, along with setting up Django Admin for user management and permissions. Custom permissions and best practices for API security, including HTTPS enforcement and rate limiting, are highlighted.

In the next chapter, we will start to build a recommendation engine, integrating complex data structures and algorithms into an API using Django.

## **Questions**

1. What is the purpose of authentication in Django?
2. What HTTP status code indicates denied authentication in Django?
3. Name one middleware used for protecting against Cross-site request forgery (CSRF) attacks in Django.
4. What does JWT stand for in the context of authentication?
5. How does Basic Authentication transmit credentials over HTTP requests?
6. Explain the role of middleware in Django.
7. How do you configure Token Authentication in Django REST Framework?

8. What is the purpose of Django's `CsrfViewMiddleware`?
9. Describe the process of adding custom permissions to Django models and integrating them with Django REST Framework.
10. Why is JWT Authentication considered stateless?

## Exercises

1. Improve the Django admin setup to include custom permissions for managing movies. Ensure that different user roles (for example, editors, administrators) have appropriate access levels.

**Task:** Create custom groups in Django admin with specific permissions for movie management.

**Hint:** Use Django's built-in permissions system to assign permissions to groups and manage user roles effectively.

2. Implement rate limiting for a specific API endpoint in your Django application to prevent abuse.

**Task:** Add rate limiting to an API view to restrict the number of requests a client can make within a defined time period.

**Hint:** Use the `django_ratelimit` package and decorate your view function with the appropriate rate limit settings.

## **CHAPTER 6**

# **Developing a Data Science Project**

## **Introduction**

This chapter is dedicated to the development of a simple recommendation engine. It introduces the foundational concepts of recommendation engines, such as collaborative filtering, content-based filtering, and hybrid methods. We will walk through the process of integrating a recommendation system into a Django web application using an API. Readers will also learn about essential data preparation techniques, environment setup using Docker, and how to implement and test recommendation algorithms.

## **Structure**

This chapter covers the following topics:

- Introduction to Recommendation Engines
- The Recommendation Algorithm
- The Role of Cosine Similarity
- Preparing the Local Environment
- Cleanup the Data
- Recommendation System Implementation Using Django
- Validating Recommendation System with Unit Tests
- Recommendation System Implementation

## **Introduction to Recommendation Engines**

A recommendation engine is a software tool to help users discover new products, content, or services that align with their interests. Recommendation engines use data on past behavior, preferences, or interactions and with this data, it tries to predict what users might find useful.

The usage of recommendation engines is very common and it is very probable that you already had recommendations when using streaming video platforms or even when shopping online.

There are different types of recommendation engines, each using a different approach to generate suggestions:

- **Collaborative Filtering:** This approach is based on the fact that people who have agreed on items in the past are likely to agree on items in the future. For instance, if two users have a history of enjoying the same movies, the system might recommend fields that one user liked but the other has not seen yet.
- **Content-Based Filtering:** This method focuses on the attributes of items rather than the behavior of users. It suggests items that are similar to ones the user has liked before. For example, if the user has enjoyed action movies in the past, the engine will recommend other action movies based on their genre, director, or actors.
- **Hybrid Methods:** Many modern recommendation systems combine collaborative filtering and content-based filtering, providing more accurate and personalized recommendations.

## [The Recommendation Algorithm](#)

We need to connect what users are interested in with what is available in the movies table. We will begin by analyzing the attributes that define each movie, such as genres, country of origin, or release year in a standardized format. Similarly, the user's preferences, which might include genres, countries or release years, are translated into the same format.

Once both movies and user preferences are represented in this comparable format, we can assess the degree of alignment between them. The closer the match between a movie's attributes and the user's preference, the more likely that our recommendation is to be good.

## [The Role of Cosine Similarity](#)

We need a way to determine the alignment between user preferences and movie attributes, for this, we will use the cosine similarity. We will measure the similarity between two vectors by focusing on the angle between them



rather than their magnitude. In our recommendation system, these vectors represent the user's preferences and the movie's attributes. The cosine similarity evaluates how closely their vectors point in the same direction.

**INFO:**

*Cosine similarity can handle data of different scales. For example, a movie might have numerous attributes, or a user might have strong preferences for certain features. Cosine similarity ensures that the comparison remains fair by focusing solely on the direction of the vectors, making it particularly effective in recommendation systems.*

*The Cosine similarity is also computationally efficient, which is important when dealing with big datasets.*

*For example, if a user has primarily watched action movies, cosine similarity doesn't limit recommendations to only that genre. Instead, it captures the direction of their interests, suggesting not only action movies but also films in similar genres, like thrillers or sci-fi, that they might enjoy.*

The result of this calculation is a similarity score ranging from -1 to 1, where 1 indicates perfect similarity, 0 indicates no similarity, and -1 indicates complete dissimilarity. Movies with the highest similarity scores will allow us to generate recommendations that are closely aligned with the user's interests, ensuring a more personalized and satisfying experience.

The recommendation we will use is a content-based filtering algorithm. It recommends items by analyzing the attributes of each item and comparing them to the user's preferences. By converting both the item attributes and user preferences into vectors and calculating their similarity—often using cosine similarity—the algorithm identifies items that closely match what the user likes. This method allows for highly personalized recommendations based on the specific characteristics of the items and the user's interests.

## [Preparing the Local Environment](#)

We will need to test our recommendation algorithm in our localhost, ideally, we should replicate our local environment as much as close to the production environment. In most of the multi-user projects, the database choice is commonly PostgreSQL. Up until now, we have been using SQLite.

SQLite use case is commonly used for embedded applications or single-user. SQLite is a great database but it has an important restriction, only one process can write to the database. If you try to upload a big file to the API and the celery worker uses a multiprocessing or multi-thread pool, you will see many errors in writing to the database.

Since we are using docker-compose, it will be easy to change our local environment.

Open the **docker-compose.yml** file and add the Postgres service:

```
services:
  ...
  postgres:
    image: postgres:latest
    environment:
      POSTGRES_USER: yourusername
      POSTGRES_PASSWORD: yourpassword
      POSTGRES_DB: yourdatabase
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
volumes:
  postgres_data:
```

In the new configuration above, a **PostgreSQL** container is defined using the **postgres:latest** image. The environment section specifies the PostgreSQL username, password, and database name, which will be used when setting up the database. The volumes section maps a local volume (**postgres\_data**) to the container's data directory (**/var/lib/postgresql/data**), for persistent storage of database files. The ports section maps port 5432 on the container to port 5432 on the host machine, allowing external access to the PostgreSQL database.

Next, we need to update the project settings, open the file **src/recommendation\_system/settings.py** and set the **DATABASES** to the value:

```
DATABASES = {
    "default": {
```

```

"ENGINE": "django.db.backends.postgresql",
"NAME": os.getenv("POSTGRES_DB", "yourdatabase"),
"USER": os.getenv("POSTGRES_USER", "yourusername"),
"PASSWORD": os.getenv("POSTGRES_PASSWORD", "yourpassword"),
"HOST": os.getenv("POSTGRES_HOST", "localhost"),
"PORT": os.getenv("POSTGRES_PORT", "5432"),
}
}

```

The **DATABASES** setting defines the connection to a PostgreSQL database. The **ENGINE** specifies the PostgreSQL backend, while the **NAME**, **USER**, **PASSWORD**, **HOST**, and **PORT** fields pull their values from environment variables (using `os.getenv`). These variables represent the database name (`POSTGRES_DB`), username (`POSTGRES_USER`), password (`POSTGRES_PASSWORD`), host (`POSTGRES_HOST`), and port (`POSTGRES_PORT`). Default values are provided if the environment variables are not set, ensuring flexibility in configuring database connections across different environments.

## Cleanup the Data

For cleaning the data, we will use the Natural Language Toolkit (NLTK). NLTK is a Python library used for natural language processing (NLP). It provides a wide range of tools and utilities for working with human language data, including tokenization, stemming, lemmatization, part-of-speech tagging, parsing, and more. NLTK also includes a vast collection of text corpora and lexical resources, such as WordNet, to support linguistic analysis.

Let us start with installing it to our project:

```

Install nltk
rye add nltk

```

Next, we need to download `nltk` required resources to your environment

Create a new script with the name `nltk_download.py`, add the following content:

```

import nltk

nltk.download('punkt')

```

```
nltk.download('wordnet')
nltk.download('stopwords')
nltk.download('punkt_tab')
```

Now execute it with python to start the download:

```
rye run python nltk_download.py
```

You should see the following script output:

```
[nltk_data] Downloading package punkt to
/Users/mandarina/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package wordnet to
/Users/mandarina/nltk_data...
[nltk_data] Downloading package stopwords to
/Users/mandarina/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt_tab to
/Users/mandarina/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
```

Now open the file **src/movies/services.py** and add two new functions, **detect\_q\_strings** and **clean\_text**:

```
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
# Function to detect strings starting with 'Q' followed by
digits
def detect_q_strings(text: str) -> list:
    """
    Detects strings that start with 'Q' followed by digits,
    useful for cleaning specific formats.
    """
    pattern = r'Q\d+'
    return re.findall(pattern, text)
```

The **detect\_q\_strings** function identifies and returns all substrings in the input text that start with the letter 'Q' followed by one or more digits, commonly used for detecting Wikidata entity IDs. It uses a regular

expression pattern (r'Q\d+') to find these patterns and returns them as a list. This function is particularly useful for extracting or cleaning Wikidata results, where entity IDs typically follow this format.

```
# Define a function for cleaning text data
def clean_text(text: str) -> str:
    """
    Cleans up the text data by removing punctuation, converting
    to lowercase,
    removing stopwords, and lemmatizing the text.
    """
    if not isinstance(text, str):
        return ""

    # Convert text to lowercase
    text = text.lower()

    # Remove any non-alphanumeric characters, keeping words and
    # digits
    text = re.sub(r'^a-zA-Z0-9\s$', '', text)

    # Tokenize the text into words
    words = word_tokenize(text)

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]

    # Initialize lemmatizer and lemmatize the words
    lemmatizer = WordNetLemmatizer()
    words = [lemmatizer.lemmatize(word) for word in words]

    # Join words back into a cleaned string
    return ' '.join(words)
```

The **clean\_text** function processes and cleans textual data for natural language processing tasks. It first converts the input text to lowercase and removes any non-alphanumeric characters. The text is then tokenized into individual words. After tokenization, common stopwords (for example, “and”, “the”) are filtered out. Each word is then lemmatized, reducing words to their base forms (for example, “running” becomes “run”). Finally,

the cleaned words are joined back into a single string. This function is useful for preparing text data for tasks like text classification or analysis by removing noise and standardizing the input.

**INFO:**

*Lemmatization is a natural language processing technique that reduces words to their base or root form, known as the “lemma.” Unlike stemming, which simply removes word endings, lemmatization considers the context and meaning of the word, ensuring the root form is valid in the language. For example, lemmatization would convert “running” to “run” and “better” to “good.” This process helps standardize text, improving the accuracy of tasks like text classification, information retrieval, or sentiment analysis, by ensuring that different forms of a word are treated as the same entity.*

Now we need to update the `parse_csv` function located in the file `src/movies/services.py`:

```
def parse_csv(file: IO[Any]) -> int:
    """
    Parses and processes a CSV file for movie data, cleaning the
    fields.
    """
    movies_processed = 0
    reader = csv.DictReader(file)
    for row in reader:
        extra_data = row.pop("extra_data").replace("'", '"')
        try:
            extra_data_dict = json.loads(extra_data)
        except json.decoder.JSONDecodeError:
            extra_data_dict = {}
        row["extra_data"] = extra_data_dict
        try:
            row["release_year"] = int(row["release_year"])
        except ValueError:
            continue

    # Clean the fields before passing to movie creation
```

```

row["title"] = clean_text(row["title"])
row["genres"] = [clean_text(genre) for genre in
row["genres"].split(', ')]
row["country"] = clean_text(row["country"])
create_or_update_movie(**row)
movies_processed += 1
return movies_processed

```

For every text data that is processed by the function, we call the **clean\_text** to prepare the data for analysis. Raw text often contains noise like punctuation, special characters, stopwords, and inconsistent capitalization, which can negatively affect the performance of machine learning models.

## [Recommendation System Implementation Using Django](#)

For our recommendation engine, we will create a new Django application that will encapsulate all the recommendation-related logic. Using this approach will maintain a clean and modular approach and it will let us reuse the code in other projects. The recommendation engine will have its API and the interface will be through a service layer that will use Pydantic models, thus decoupling the solution to the movie models we developed in previous chapters.

Create a new application by running the **startapp**:

```
rye run python manage.py startapp recommendations
```

In the recommendations application directory, create a new file **src/recommendations/services.py**. In this file, we will create two new classes and a function **get\_recommendations** that will not be related to movies and it could be used for any other items, like books, cars, or more. For handling this communication to the service we will define some pydantic models, which are Python classes that inherit from the pydantic **BaseModel**.

Let us start with installing **pydantic**:

```
rye add pydantic
```

**INFO:**

*Pydantic is a powerful data validation and parsing library in Python, designed to make data handling easy and reliable. It uses Python's type hints to validate, serialize, and deserialize data, ensuring your data models are well-structured and type-safe. By automatically converting input data to the correct types and enforcing constraints, Pydantic helps reduce errors and improve code quality, making it an essential tool for building robust and maintainable applications.*

Add the two new pydantic models:

```
from pydantic import BaseModel

class Item(BaseModel):
    id: int
    attributes: dict

class UserPreferences(BaseModel):
    preferences: dict | None = None
    watch_history: list[int] | None = []
```

These two Pydantic models will be used as input for the recommendation system. **Item** holds an id and attributes as a flexible dictionary for metadata like **genres** or **actors**. **UserPreferences** tracks a user's optional **preferences** and their **watch\_history** as a list of item IDs. These models provide a clean, flexible structure to handle item data and user interactions.

The interface for our **get\_recommendations** function will be:

```
def get_recommendations(user_preferences: UserPreferences,
                        items: list[Item], top_n: int = 10) -> list[Item]:
    pass
```

To initialize the Pydantic models and call the **get\_recommendations** function, you first need to create instances of **Item** and **UserPreferences**. For example, you can initialize an **Item** with an id and an attributes dictionary that holds metadata such as genres or actors. Similarly, the **UserPreferences** model can be initialized with a dictionary of preferences and a list of previously watched items. Once the models are created, you pass them to the **get\_recommendations** function along with the list of items and specify the number of top recommendations you would like to retrieve.

**Example:**



```

items = [
    Item(id=1, attributes={"title": "Midnight Express",
        "genre": "Crime, Drama",
        "actor": "Brad Davis",
        "director": "Alan Parker",
        "year": 1978}),
    Item(id=2, attributes={"title": "Blue Velvet",
        "genre": "Mystery, Thriller",
        "actor": "Kyle MacLachlan",
        "director": "David Lynch",
        "year": 1986})
]

user_preferences = UserPreferences(
    preferences={"genre": "Crime, Drama",
        "actor": "Brad Davis",
        "director": "Alan Parker",
        "year": 1978},
    watch_history=[1] # The user has already watched Midnight
    Express
)

recommended_items = get_recommendations(user_preferences,
items, top_n=5)

```

Here, the `get_recommendations` function will process the user's preferences and watch history against the list of items to return the top 5 recommendations.

## [Validating Recommendation System with Unit Tests](#)

Writing tests ensures the functionality of the code and prevents regression. When writing tests for a recommendation system it is important to distinguish between tests designed to detect functional issues (breaking changes) and those that evaluate the quality of recommendations. Following the TDD methodology, we will start with tests:

```

def test_top_3_recommendations_with_more_noise(
    sample_items, user_preferences

```

```

):
    recommendations = get_recommendations(
        user_preferences, sample_items, top_n=3
    )

    # The top 3 recommendations should include highly relevant
    items but also account for similar Sci-Fi items like Blade
    Runner
    expected_items = {"Aliens", "True Lies", "Predator"}
    acceptable_alternatives = {
        "Blade Runner"
    } # Blade Runner is also relevant due to Sci-Fi genre

    # Extract the names of the top 3 recommendations
    recommended_item_names = {rec.name for rec in
    recommendations}

    # Ensure that all expected or acceptable items are in the top
    recommendations
    assert (expected_items & recommended_item_names) or (
        acceptable_alternatives & recommended_item_names
    ), (
        f"Top recommendations should include: {expected_items} or
        alternatives like {acceptable_alternatives}, "
        f"but got: {recommended_item_names}"
    )

```

This test case ensures that given specific user preferences (for example, for “Action” genre, “Arnold Schwarzenegger” as an actor, “James Cameron” as a director, and specific years), the top three recommendations include relevant items like Aliens, Predator, and True Lies. In addition, acceptable alternatives (such as Blade Runner due to its genre overlap) are also considered.

This test ensures that relevant items, based on the user’s input preferences, are included in the recommendations. If a breaking change in the code prevents these items from appearing, the test will fail, alerting developers to a functional problem.

While this test ensures that the algorithm returns relevant items, it doesn’t measure the quality of those recommendations in terms of ranking or

overall user satisfaction.

```
def test_recommendations_handle_more_noise(sample_items,
user_preferences_with_noise):
    recommendations = get_recommendations(
        user_preferences_with_noise, sample_items, top_n=5
    )
    noisy_items = {
        "Sleepless in Seattle",
        "Pride and Prejudice",
        "The Notebook",
        "La La Land",
        "Casablanca",
    }

    for item in recommendations:
        assert item.name not in noisy_items, f"Noisy item
        '{item.name}' should not be in recommendations"
```

This test ensures that items irrelevant to the user's preferences (for example romance or musical films) are excluded from the recommendations. It verifies that the recommendation system respects the user's preferences by not recommending genres or movies the user is unlikely to enjoy.

It detects when unwanted recommendations (for example, genres or actors not matching the user's preferences) slip into the results. If, for example, the system suggests *Sleepless in Seattle* to a user who prefers action movies, the test will catch that failure.

However, this test only confirms that noisy items are not present—it doesn't evaluate whether the system ranks the most relevant items highest, nor does it assess how closely the system tailors recommendations to user preferences.

Functional tests like these are essential for ensuring the integrity of the recommendation system. They help detect breakages and regressions, ensuring that relevant items are recommended and irrelevant ones are excluded.

To assess recommendation quality, more advanced techniques, such as A/B testing, precision and recall measurements, and user satisfaction analysis,

are required. Functional tests ensure that the system works; evaluating quality requires a deeper understanding of how well it works for real users.

**INFO:**

*A/B testing is an experimental method used to compare two versions of a product or feature to determine which one performs better. In A/B testing, users are randomly divided into two groups:*

- **Group A:** *Sees the original version (the “control”).*
- **Group B:** *Sees a modified version (the “variation”).*

By measuring user behavior or engagement (for example, clicks, conversions, or time spent), you can determine which version is more effective at achieving a specific goal. A/B testing helps optimize features, designs, or algorithms by using real-world user data to make informed decisions.

## [Recommendation System Implementation](#)

The algorithm to calculate the recommendations will use scikit-learn, let us install it with **rye**:

```
rye add scikit-learn
```

**INFO:**

*Scikit-learn is a powerful and widely-used open-source machine learning library in Python, designed to simplify the implementation of a broad range of machine learning algorithms. It provides efficient tools for data analysis and modeling, offering functionalities for classification, regression, clustering, dimensionality reduction, and preprocessing. Built on top of NumPy, SciPy, and Matplotlib, scikit-learn is well-integrated within the broader scientific Python ecosystem, making it easy to handle complex data structures and visualize results. Its user-friendly API, extensive documentation, and robust community support make it an ideal choice for both beginners and experienced practitioners in data science, allowing them to rapidly prototype and deploy machine learning models in real-world applications.*

We are ready to implement the **get\_recommendations** function. The implementation will use content-based filtering:

```
from sklearn.metrics.pairwise import cosine_similarity
def get_recommendations(user_preferences: UserPreferences,
items: list[Item], top_n: int = 10) -> list[Item]:
    # Combine item attributes into a single text field for
    vectorization
    combined_tags = [combine_attributes(item) for item in items]

    # Initialize a CountVectorizer
    cv = CountVectorizer(max_features=10000,
stop_words='english')

    # Combine user preferences into a single text field
    raw_user_tags = []
    if user_preferences.preferences:
    for key, value in user_preferences.preferences.items():
        if isinstance(value, list):
            raw_user_tags += [str(v).lower() for v in value]
        else:
            raw_user_tags.append(str(value).lower())

    user_tags = " ".join(raw_user_tags)

    # Fit CountVectorizer on both the items and user preferences
    cv.fit(combined_tags + [user_tags])

    # Vectorize the combined tags and user preferences
    item_vectors = cv.transform(combined_tags).toarray()
    user_vector = cv.transform([user_tags]).toarray()

    # Calculate cosine similarity between the user preferences
    and all item vectors
    similarity_scores = cosine_similarity(user_vector,
item_vectors)[0]

    # Rank items by similarity score and get top_n items
    ranked_items = sorted(enumerate(similarity_scores),
key=lambda x: x[1], reverse=True)

    # Exclude items the user has already interacted with
```

```
ranked_items = [items[i] for i, score in ranked_items if
items[i].id not in user_preferences.watch_history]

# Return the top_n items
return ranked_items[:top_n]
```

The function relies on a **CountVectorizer** to transform textual data into vectors, followed by cosine similarity to rank items based on relevance to the user's preferences.

#### **INFO:**

**CountVectorizer** is a scikit-learn tool that transforms text data into numerical vectors by tokenizing the text and counting the occurrences of each word. It creates a matrix where each unique word becomes a feature, and the frequency of that word in the text is recorded. In a recommendation system, it is used to convert both item attributes and user preferences into vectors, enabling numerical comparisons, such as calculating the similarity between items and preferences.

First, we process each Item by combining its attributes into a single text field, which serves as the input for vectorization. The **CountVectorizer** is initialized to tokenize and remove common stop words from these fields. User preferences are also combined into a single string, ensuring that each value is converted to lowercase and properly formatted for vectorization.

The **CountVectorizer** is then trained on both the item attributes and the user preferences. After training, we transform the items and user preferences into numerical vectors. Using these vectors, we calculate cosine similarity, a measure of how closely related the user preferences are to each item's attributes.

#### **INFO:**

*Cosine similarity is commonly used in recommendation systems because it measures the similarity between two vectors based on their orientation, rather than their magnitude. This makes it ideal for comparing text data, like user preferences and item attributes, which are often converted into vectors of word counts. By focusing on the angle between vectors, cosine similarity provides a scale from 0 (no similarity) to 1 (perfect similarity).*

The similarity scores are sorted, and the items are ranked accordingly. To ensure meaningful recommendations, we filter out any items that the user has already interacted with, using the **watch\_history** from **UserPreferences**. Finally, the function returns the top N recommendations based on the highest similarity scores.

Finally, we have the helper function **combine\_attributes**:

```
def combine_attributes(item: Item) -> str:
    """
    Combines the attributes of an item into a single string for
    vectorization.
    If an attribute is a list (e.g., genres), it flattens the
    list into a space-separated string.
    """
    combined_attributes = []

    for value in item.attributes.values():
        if isinstance(value, list):
            # Flatten the list into a space-separated string
            combined_attributes.append(" ".join(v.lower() for v in
            value))
        else:
            # Convert the value to a string and make it lowercase
            combined_attributes.append(str(value).lower())

    return " ".join(combined_attributes).strip()
```

The **combine\_attributes** function merges an item's attributes into a single lowercase string for vectorization. If an attribute is a list (for example, genres), it flattens it into a space-separated string; otherwise, it converts the value to lowercase. This unified string format allows for easy text processing and similarity calculations. For example, an item with **{"genre": ["Film based on book", "Prison film"], "actor": "Brad Davis"}** becomes **"film based on book prison film brad davis"**.

Now that we know how to use the recommendation system, we need to add an API to calculate movie recommendations for our users. Since we want our recommendation app to be reusable for other models, such as books, products or any other item we will add the API to get the movie

recommendation in the movies application. The movie application will import the **get\_recommendation** service function to make the calculations.

Open the file **src/movies/api.py** and add the new class to implement the new GET API that will return the recommendations:

```
class MovieRecommendationAPIView(APIView):
    def get(self, request):
        user_id = request.query_params.get("user_id")

        if not user_id:
            return self._response_error(
                detail="user_id query parameter is required.",
                status_code=status.HTTP_400_BAD_REQUEST,
            )

        user_preferences = self._get_user_preferences(user_id)
        if not user_preferences:
            return self._response_error(
                detail="User preferences not found.",
                status_code=status.HTTP_404_NOT_FOUND,
            )

        recommended_items =
            self._get_recommended_items(user_preferences)
        response_data = self._format_response(recommended_items)

        return Response(response_data, status=status.HTTP_200_OK)
```

The **MovieRecommendationAPIView** class defines an API for retrieving movie recommendations based on user preferences. The `get` method checks for a valid **user\_id** in the query parameters, retrieves the user's preferences using **\_get\_user\_preferences**, and generates movie recommendations through **\_get\_recommended\_items**. The results are formatted in **\_format\_response** and returned as a JSON response. If the **user\_id** is missing or preferences are not found, an error response is generated using **\_response\_error**. The core logic involves fetching movie data, structuring it into Item objects, and calling the **get\_recommendations** function to return personalized recommendations for the user.

```
def _get_user_preferences(self, user_id: str) ->
    Optional[UserPreferences]:
```



```

"""
Retrieves user preferences from the database and converts
them into the UserPreferences Pydantic model.
:param user_id: The ID of the user whose preferences are to
be fetched.
:return: A UserPreferences object populated with the user's
preferences, or None if no preferences exist.
"""
try:
    # Fetch user preferences from the database
    user_prefs =
    UserMoviePreferences.objects.get(user_id=user_id)

    # Extracting the relevant preferences
    genre = user_prefs.preferences.get("genre", [])
    director = user_prefs.preferences.get("director", [])
    actor = user_prefs.preferences.get("actor", [])
    year_range = user_prefs.preferences.get("year", [])

    # Handle year range if it's provided (expecting a list)
    year_range_start, year_range_end = (
        (year_range[0], year_range[-1]) if len(year_range) >= 2
        else (None, None)
    )

    # Build the preferences dictionary
    preferences_dict = {
        "genre": genre,
        "director": director,
        "actor": actor,
        "year_range_start": year_range_start,
        "year_range_end": year_range_end,
    }

    # Instantiate UserPreferences with the preferences
    dictionary
    return UserPreferences(
        preferences=preferences_dict,

```

```

        watch_history=user_prefs.watch_history, # Assuming this
        exists in user_prefs
    )
except UserMoviePreferences.DoesNotExist:
    return None # Return None if no preferences are found for
    the user

```

The **\_get\_user\_preferences** method retrieves a user's movie preferences from the database and converts them into the **UserPreferences** Pydantic model. It fetches the user's preferences, extracting relevant fields like genre, director, actor, and year range. If a year range exists, it is split into **year\_range\_start** and **year\_range\_end**. These values are compiled into a preferences dictionary, which, along with the user's **watch\_history**, is used to create a **UserPreferences** object. If no preferences are found for the user, the method returns None.

```

def _get_recommended_items(self, user_preferences:
UserPreferences) -> list[Item]:
    """
    Generates a list of recommended items (in this case, movies)
    based on user preferences.

    :param user_preferences: The preferences of the user.
    :return: A list of recommended items as Item objects.
    """
    movies = Movie.objects.all()

    items = [
        Item(
            id=movie.id,
            attributes={
                "name": movie.title,
                "genre": movie.genres,
                "director": movie.extra_data.get("directors", ""),
                "year": movie.release_year,
            }
        )
        for movie in movies
    ]

```

```
    return get_recommendations(user_preferences=user_preferences,
                               items=items)
```

The `_get_recommended_items` method generates a list of recommended movies based on the user's preferences. It retrieves all movies from the database, and for each movie, it creates an `Item` object with attributes like title, genres, director, and release year. These items are then passed, along with the user's preferences, to the `get_recommendations` function, which returns a list of recommended movies. The method structures the data in a way that makes it ready for recommendation processing.

To make our API available, we need to add it to the `urls.py` of the movies application, open the file `src/movies/urls.py` and add the new pattern:

```
from movies.api import (
    ...
    MovieRecommendationAPIViewSet,
)
urlpatterns = [
    ...
    path("recommendations/",
         MovieRecommendationAPIViewSet.as_view(),
         name="movie_recommendations")
]
```

Now we will test our recommendations system, for this we will first upload the movies using the CSV file we generated in [Chapter 3, Data Models and Processing in Data Science](#).

The first step is to get the authentication token using our auth API:

```
$ curl -X POST -H "Content-Type: application/json" \
  -d '{"username": "mandarina", "password": "SECUREPASSWORD!"}' \
  -w "\nStatus: %{http_code}\n" \
  http://localhost:8000/auth/token/
{"token": "0d92eadc39e6e1d857caecdbdd6148ade5424e92"}
Status: 200
```

Using this token we will upload the movies csv file:

```
curl -X POST -H "Authorization: Token
0d92eadc39e6e1d857caecdbdd6148ade5424e92" \
  -H "Content-Type: multipart/form-data" \
```

```
-F "file=@all_movies_data.csv;type=text/csv" \
-w "\nStatus: %{http_code}\n" \
http://localhost:8000/api/v1/movies/upload/
```

Now we will simulate some users' preferences, first we need to create a new user. Go to the admin users page (<http://localhost:8000/admin/auth/user/add/>) and create a new one:

The screenshot shows the Django administration interface. The left sidebar has a search bar and several sections: 'AUTH TOKEN' with a 'Tokens' link; 'AUTHENTICATION AND AUTHORIZATION' with 'Groups' and 'Users' links; and 'MOVIES' with 'Movies' and 'User movie preferences' links. The 'Users' link is highlighted. The main content area is titled 'Add user' and contains the following text: 'First, enter a username and password. Then, you'll be able to edit more user options.' Below this are three input fields: 'Username:', 'Password:', and 'Password confirmation:'. The 'Username' field has a note: 'Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only.' The 'Password' field has three notes: 'Your password can't be too similar to your other personal information.', 'Your password must contain at least 8 characters.', and 'Your password can't be a commonly used password.' The 'Password confirmation' field has a note: 'Enter the same password as before, for verification.' At the bottom of the form are three buttons: 'SAVE', 'Save and add another', and 'Save and continue editing'.

**Figure 6.1:** Creating a user for testing recommendations

Once you create the user, the admin will redirect to the user edit page. This page will have a user ID in the url.

With this user ID, we will use the API to add some movie preferences:

```
curl -X POST -H "Authorization: Token
0d92eadc39e6e1d857caecdbdd6148ade5424e92" \
-H "Content-Type: application/json" \
-d '{
  "new_preferences": {
    "genre": "film based on book",
    "director": "Alan Parker",
    "actor": "Brad Davis",
```

```
    "year": 1978
  }
}' \
-w "\nStatus: %{http_code}\n" \
http://localhost:8000/api/v1/users/<USER_ID>/preferences/
```

Now, we are ready to get the recommendation using the API:

```
curl -X GET -H "Authorization: Token
19393fd8d8bb9e2f5695ba0be9ab1da50f847d8f" \
-w "\nStatus: %{http_code}\n" \
"http://localhost:8000/api/v1/movies/recommendations/?
user_id=1"
```

The result obtained was:

```
[{"id":10262,"title":"midnight express"},
{"id":119456,"title":"city lie"},
{"id":58353,"title":"serpico"},
{"id":79059,"title":"goodfellas"}, {"id":89071,"title":"donnie
brasco"}]
```

We can see that the recommendation system delivers a strong match with Midnight Express, which aligns perfectly with the user's preferences in terms of year, director, and genre. However, the other recommendations, such as Serpico, Goodfellas, and Donnie Brasco, while thematically relevant as crime and drama films, fall short in meeting the user's specific criteria for actors and year. This demonstrates that while the system is capable of producing useful results. The current model could benefit from refining the weighting of preferences like actor and year, ensuring a more precise alignment with the user's taste. Nonetheless, this serves as a solid foundation, showing the system's potential to deliver valuable recommendations with further tuning.

## Conclusion

In this chapter, we built a content-based recommendation engine using cosine similarity, walking through the entire process from concept to implementation. We introduced key concepts like collaborative filtering, content-based filtering, and hybrid methods. The focus was on integrating

the engine into a Django application, using Docker and PostgreSQL for setup, and employing NLTK for data cleaning.

We developed reusable Pydantic models and implemented a modular **get\_recommendations** function. We also integrated this with a Django API and validated the system through unit tests, ensuring accurate results.

While the system performed well, we noted areas for improvement, such as refining preference weighting. This project provides a strong foundation for building scalable, flexible recommendation systems that can be further enhanced for greater accuracy and personalization.

In the next chapter, *Documenting and Optimizing Your Data API*, we will learn how to make great documentation for our API and why it is important to have it. We will also learn some ideas on how to optimize our project.

## Questions

1. What is a recommendation engine, and why is it commonly used?
2. What is the main difference between collaborative filtering and content-based filtering?
3. What is cosine similarity, and how is it used in recommendation systems?
4. Why do we use PostgreSQL instead of SQLite in multi-user environments?
5. How does the **clean\_text** function help prepare data for a recommendation system?

## Exercises

1. Implement Data Cleaning for Reviews: Modify the **clean\_text** function to clean user reviews, including removing HTML tags, URLs, and emojis.
2. Add New Attributes to the Recommendation System: Extend the **get\_recommendations** function by adding a new attribute, such as "rating" or "language," and use it in the recommendation calculations.

[OceanofPDF.com](http://OceanofPDF.com)

# CHAPTER 7

## Documenting and Optimizing Your API

### Introduction

In today's API-driven development landscape, documentation is crucial for both developers and users. It acts as the contract for integrating an API into applications. This chapter explores why documenting your API with OpenAPI is essential and provides guidance on how to implement it in Django Rest Framework (DRF). Additionally, we will delve into optimizing your API's performance, covering crucial techniques like caching, reducing payload sizes, and implementing rate limiting and throttling to safeguard your application. By the end of this chapter, you will have the tools to create robust documentation and optimize your API for scalability and efficiency.

### Structure

This chapter covers the following topics:

- Introduction to OpenAPI
- Setting Up OpenAPI in Django Rest Framework
- Documenting API Endpoints
- Performance Optimization Techniques for APIs

### Introduction to OpenAPI

OpenAPI started as the Swagger Specification in 2010 to standardize API documentation. Thanks to its ease of use and ability to auto-generate documentation, it quickly gained popularity.



In 2015, the Swagger specification was renamed to OpenAPI and it became a project of the OpenAPI initiative. The OpenAPI initiative aims to provide a formal and standardized specification for APIs.

Today, OpenAPI is one of the most widely adopted standards for documenting APIs.

## Understanding OpenAPI

The OpenAPI Specification is a language-agnostic format for describing APIs. OpenAPI allows you to describe the entire API, including:

- **Path:** Each API endpoint (example: /movies) is defined in the OpenAPI documents.
- **Methods:** The HTTP methods are supported by each endpoint.
- **Parameters:** Query parameters, path variables and request bodies.
- **Responses:** The structure of the response, including status code and errors.
- **Security:** How your API handles authentication and authorization.

The API documentation serves as the single source of truth. It not only provides developers with information on how to use the API, but also allows them to generate clients, create test cases, and use tools like Postman. Moreover, the API documentation acts as a contract, ensuring that our API users can reliably interact with the system, making it an essential part of the development process.

## Using OpenAPI

While Django Rest Framework (DRF) is a powerful tool for building RESTful APIs, it doesn't provide interactive documentation out of the box. We will need to install a third-party library to be able to generate the API documentation. Without additional libraries, manually writing and maintaining API documentation becomes tedious and error-prone.

When we integrate OpenAPI into our project, we can:

- **Auto Generate Documentation:** OpenAPI extracts endpoint details, methods and parameters from the code, and the documentation is automatically generated.

- **Keep Documentation Updated:** Since the documentation is generated from the code, it stays up to date.
- **Interactive Documentation:** The generated documentation allows anyone with a browser to interact and use the API in a friendly way.
- **Reduce Support Inquiries:** Since our API documentation will contain many details, it will reduce the back and forth with our API users. The documentation should answer most of their questions.

## Setting Up OpenAPI with drf-spectacular

First, we need to install the necessary library to integrate OpenAPI with DRF. One of the most popular libraries for this is **drf-spectacular**, which simplifies the process of generating OpenAPI documentation.

Run the following command:

```
rye add drf-spectacular
```

The next step is to configure our project to generate the documentation. Open the `settings.py` file located in `src/recommendation_system/settings.py` and add the following configuration:

```
INSTALLED_APPS = [  
    ...  
    "drf_spectacular",  
]  
  
REST_FRAMEWORK = {  
    ...  
    "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema",  
}  
  
SPECTACULAR_SETTINGS = {  
    "TITLE": "Movie Recommendation API",  
    "DESCRIPTION": "An API for managing movies, user preferences,  
    and recommendations.",  
    "VERSION": "1.0.0",  
    "SERVE_INCLUDE_SCHEMA": False,  
}
```

First, add `drf_spectacular` to the `INSTALLED_APPS` to enable DRF Spectacular.

The `REST_FRAMEWORK` configuration specifies using `drf-spectacular`'s `AutoSchema` as the default schema class, which automatically generates schema definitions for API endpoints. The `SPECTACULAR_SETTINGS` dictionary customizes the generated OpenAPI documentation, including the title ("**Movie Recommendation API**"), a brief description of the API's purpose, the version of the API, and a flag (`SERVE_INCLUDE_SCHEMA: False`) which prevents the schema from being served as part of the API itself.

### **Adding URL Endpoints for API Documentation:**

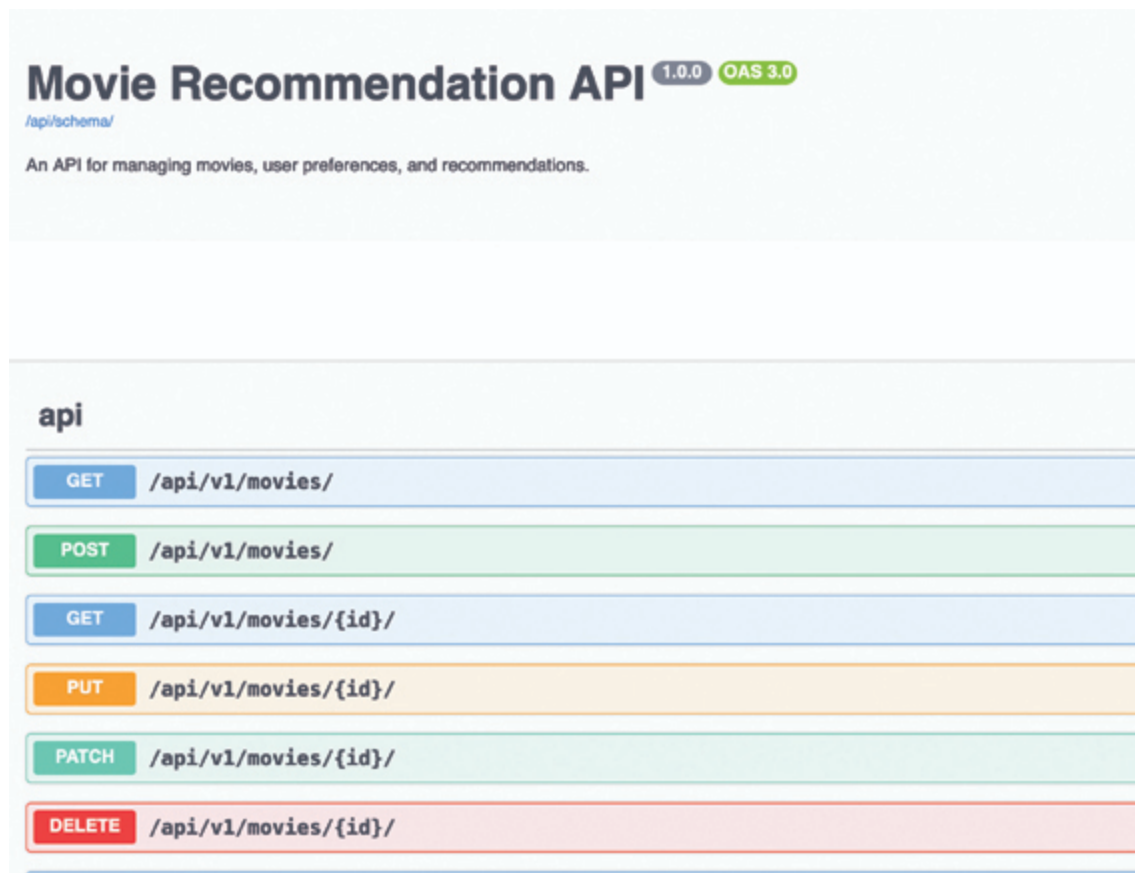
Open the `urls.py` of the project located at `src/recommendation_system` and add the following URLs:

```
from drf_spectacular.views import SpectacularAPIView,
SpectacularSwaggerView, SpectacularRedocView

urlpatterns = [
    ...
    path("api/schema/", SpectacularAPIView.as_view(),
        name="schema"),
    path("api/docs/",
        SpectacularSwaggerView.as_view(url_name="schema"),
        name="swagger-ui"),
    path("api/redoc/",
        SpectacularRedocView.as_view(url_name="schema"),
        name="redoc"),
    path("api/v1/movies/", include("movies.urls")),
]
```

- **/api/schema/**: Provides the raw OpenAPI schema in JSON format.
- **/api/docs/**: Renders the interactive Swagger UI, allowing developers to test API endpoints.
- **/api/redoc/**: Renders documentation using the Redoc UI, another popular documentation interface.

Now, in your browser and open the link `http://localhost:8000/api/docs/`, you should see:



*Figure 7.1: Movies API documentation*

At this point, you should see fully interactive, auto-generated documentation for your API, including all endpoints, request parameters, and response formats. You can test your API directly from the browser using these interfaces.

## [Documenting API Endpoints](#)

Now that we have set up OpenAPI using **drf-spectacular**, we are ready to improve the existing endpoints of our movies API.

Open the file `src/movies/api.py` and decorate the `MovieListCreateAPIView` using drf-spectacular `extend_schema`:

```
from drf_spectacular.utils import extend_schema,
OpenApiParameter

@extend_schema(
    summary="Retrieve all movies",
```

```

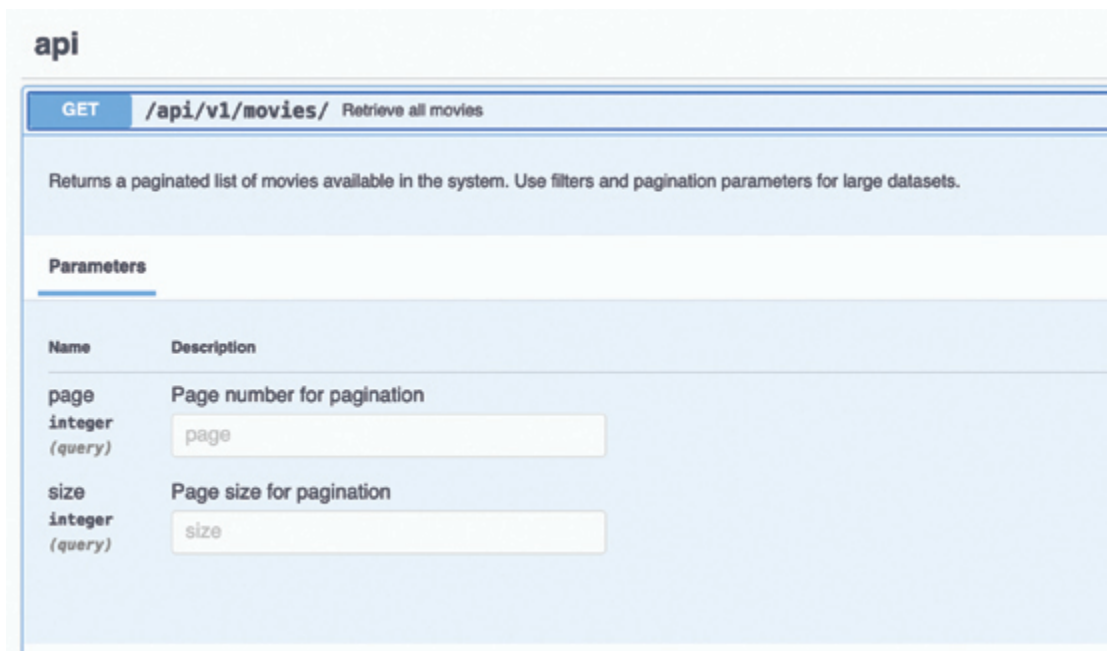
description="Returns a paginated list of movies available in
the system. Use filters and pagination parameters for large
datasets.",
responses={
    200: MovieSerializer(many=True), # Response schema when
    successful
},
parameters=[
    OpenApiParameter("page", int, description="Page number for
    pagination"),
    OpenApiParameter("size", int, description="Page size for
    pagination"),
],
methods=["GET"], # Explicitly document GET
)
class MovieListCreateAPIView(generics.ListCreateAPIView):
    ...

```

Using the `@extend_schema` decorator from the **drf-spectacular** package to enhance the API documentation for the **MovieListCreateAPIView**, which handles listing and creating movies. The decorator adds metadata to the OpenAPI schema for the endpoint, providing a clear and structured API description. The **summary** gives a brief overview (“Retrieve all movies”), and the **description** explains that the API returns a paginated list of movies, allowing users to apply filters and pagination parameters for handling large datasets. The **responses** section specifies the expected response schema, using the **MovieSerializer** to represent a successful response (HTTP 200) with multiple movie objects. Additionally, the **parameters** section defines the query parameters for pagination, such as page and size, each with a description to inform users how to paginate through large datasets.

Since **MovieListCreateAPIView** handles both listing and creating movies, we explicitly document the GET method for listing.

If you now re-open the API documentation [http://localhost:8000/api/docs/#/api/api\\_v1\\_movies\\_list](http://localhost:8000/api/docs/#/api/api_v1_movies_list), you will find more information shown in the generated documentation:



*Figure 7.2: Additional information in the API documentation*

Let us add the documentation for the creation endpoint, just below the previous decorator, add a new one to document the POST method:

```
from drf_spectacular.utils import extend_schema,
OpenApiParameter, OpenApiResponse

@extend_schema(
    ...
    methods=["GET"], # Explicitly document GET
)
@extend_schema(
    summary="Create a new movie",
    description="Adds a new movie to the system. Requires
authentication and appropriate permissions.",
    request=MovieSerializer, # Request body schema for POST
    responses={
        201: MovieSerializer, # Successful creation response schema
        400: OpenApiResponse(description="Bad Request. Validation
error."),
        403: OpenApiResponse(description="Forbidden. Insufficient
permissions."),
    },
),
```

```

    methods=["POST"], # Explicitly document POST
)
class MovieListCreateAPIView(generics.ListCreateAPIView):
    ...

```

In the provided code snippet, the `@extend_schema` decorator is used to extend the documentation of the **POST** method of the **MovieListCreateAPIView** class, which is responsible for creating new movies. The **summary** and **description** fields provide a clear overview of the API's functionality. The **request=MovieSerializer** specifies the expected format of the data being sent in the request body, which is validated using the **MovieSerializer** class. The **responses** field defines potential responses, such as a 201 status code for a successful creation or error responses like 400 for invalid input and 403 for insufficient permissions. By explicitly specifying **methods=["POST"]**, the decorator ensures that this schema applies only to the POST operation.

Similarly, we can document the **MovieDetailAPIView** by adding the decorator to the class:

```

@extend_schema(
    summary="Update a movie by ID",
    description="Updates the details of an existing movie. Requires authentication and proper permissions.",
    request=MovieSerializer, # Request body schema
    responses={
        200: MovieSerializer, # Success response
        400: OpenApiResponse(description="Bad Request. Validation error."),
        403: OpenApiResponse(description="Forbidden. Insufficient permissions."),
        404: OpenApiResponse(description="Movie not found.")
    },
    parameters=[
        OpenApiParameter("id", int, description="ID of the movie to update"),
    ],
    methods=["GET"],
)

```

```
class
MovieDetailAPIView(generics.RetrieveUpdateDestroyAPIView):
    ...
```

Again, we use the `@extend_schema` decorator to provide API documentation for updating a movie by ID. It includes a **summary** and **description**, noting that authentication and permissions are required. The request body is represented by **MovieSerializer**, and responses include success (200), validation errors (400), insufficient permissions (403), and movie not found (404). The id of the movie is specified as a path parameter. We also set the **methods** to only GET since the class implements the PUT and DELETE methods.

As a final example, we will add documentation to the file upload endpoint:

```
@extend_schema(
    summary="Upload a CSV or JSON file for background
processing",
    description=(
        "Uploads a CSV or JSON file and enqueues a background task
        for processing the file. "
        "The user must be authenticated. The file is saved with a
        unique name, "
        "and the task is asynchronously processed. Supported file
        types: CSV and JSON."
    ),
    request=GeneralFileUploadSerializer, # Expected input for
the file upload
    responses={
        202: OpenApiResponse(
            description="File uploaded successfully. Job enqueued for
background processing.",
            examples={"application/json": {"message": "Job enqueued for
processing."}}
        ),
        400: OpenApiResponse(description="Bad Request. Validation
error or unsupported file type."),
    },
    parameters=[
```



```
    OpenApiParameter(name="file", type="file", description="The
    CSV or JSON file to upload"),
],
methods=["POST"], # Explicitly documents the POST method
)
...
```

The **summary** and **description** clarify that authenticated users can upload a file, which is saved with a unique name and processed asynchronously. Only CSV and JSON file types are supported. The **request** format is handled by the **GeneralFileUploadSerializer**, and the **responses** include a 202 status for successful uploads, indicating the job was queued for processing, and a 400 status for validation errors or unsupported file types. The **parameters** specify that the file can be either CSV or JSON.

## [Performance Optimization Techniques for APIs](#)

Performance optimization techniques help reduce response times, minimize resource consumption, and improve the overall user experience. These techniques involve strategies such as data caching, reducing payload size, rate limiting, and optimizing query performance. By implementing these practices, developers can enhance API responsiveness, handle higher traffic loads, and ensure that their systems scale smoothly as demand grows.

## [Data Caching Techniques](#)

Caching is a technique used in software development to improve the speed and scalability of applications. Caching involves storing frequently accessed data in temporary storage to enable faster retrieval. Cache storage is typically located in memory, close to the application for faster data retrieval.

Redis is a popular in-memory data structure store often used as a cache to optimize application performance. Unlike traditional databases, Redis stores data in memory, making it extremely fast for read and write operations. Other caching solutions, such as Memcached, also serve similar purposes by temporarily storing frequently accessed data to reduce database load and improve application performance. Memcached is known for its simplicity and speed, particularly when dealing with small chunks of arbitrary data.

To set up Redis with Django, first, install the necessary package:

```
rye add django-redis
```

Configure Redis in your `src/recommendation_system/settings.py`:

```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    }
}
```

This code defines the caching configuration for a Django application using Redis as the backend. It uses the **django-redis** library to connect Django's cache framework to a Redis server. The default cache backend is specified, meaning that if no specific cache is mentioned elsewhere in the application, this configuration will be used. The **BACKEND** key points to **django\_redis.cache.RedisCache**, which tells Django to use Redis for caching. The **LOCATION** specifies where Redis is running—here it is set to `redis://127.0.0.1:6379/1`, indicating that Redis is hosted locally (localhost) on the default Redis port (6379), with the cache being stored in Redis database number 1. This setup is common for development environments, where Redis is often run locally for simplicity. In the **OPTIONS** section, the **CLIENT\_CLASS** is set to **django\_redis.client.DefaultClient**, specifies the default client class used by Django to interact with Redis. Redis has been configured with Docker Compose to work not only with Django for caching but also as the broker for Celery, which manages background tasks. This integration allows Redis to handle both caching and task queue management efficiently within the application's development environment.

You can cache entire API responses using the **@cache\_page** decorator. This can significantly reduce response times for endpoints that don't need real-time data updates.

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15) # Cache for 15 minutes
def my_view(request):
```

```
# Your view logic
```

To cache a class-based view (CBV) in Django, you can use the **`django.views.decorators.cache.cache_page`** decorator in combination with Django's **`method_decorator`**. Here is an example:

```
from django.utils.decorators import method_decorator
from django.views.decorators.cache import cache_page
from django.views.generic import TemplateView

# Example class-based view
class MyView(TemplateView):
    template_name = "my_template.html"

    @method_decorator(cache_page(60 * 15)) # Cache for 15
    minutes
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
```

This code applies caching to a Django class-based view using **`cache_page`**. The **`MyView`** class, inheriting from **`TemplateView`**, caches the view's response for 15 minutes by wrapping the `dispatch` method with **`@method_decorator(cache_page(60 * 15))`**. This ensures that all incoming HTTP requests (GET, POST, and more) are cached, improving performance by serving cached responses during that period instead of regenerating them for every request.

Caching in DRF can improve response times and reduce database load, but it is essential to choose the right cache expiration settings to ensure your data remains fresh.

## [Reducing Payload Size](#)

Reducing the payload size can improve response times and make the API more efficient, especially for large datasets. DRF provides several features to help you optimize payload size:

DRF provides built-in support for pagination. Instead of returning all results in one response, you can split the data into smaller, manageable chunks. You can configure pagination in **`settings.py`**:

```
REST_FRAMEWORK = {
```

```
'DEFAULT_PAGINATION_CLASS':  
'rest_framework.pagination.PageNumberPagination',  
'PAGE_SIZE': 10  
}
```

DRF serializers allow you to specify which fields to include in a response. You can dynamically control the fields sent to the client based on request parameters, reducing unnecessary data transfer.

Django supports response compression out of the box. GZIP compression reduces the size of responses by adding the following to your **settings.py**.

```
MIDDLEWARE = [  
    'django.middleware.gzip.GZipMiddleware',  
    # Other middleware...  
]
```

## [Rate Limiting and Throttling](#)

To safeguard your API from overuse or abuse, Django Rest Framework provides robust rate limiting and throttling mechanisms. These ensure that API users cannot overload your system with excessive requests.

Then, enable throttling in your **settings.py**:

```
REST_FRAMEWORK = {  
    'DEFAULT_THROTTLE_CLASSES': [  
        'rest_framework.throttling.UserRateThrottle',  
        'rest_framework.throttling.AnonRateThrottle',  
    ],  
    'DEFAULT_THROTTLE_RATES': {  
        'user': '1000/day', # Custom rate for authenticated users  
        'anon': '100/day' # Custom rate for anonymous users  
    }  
}
```

IP-based throttling is an effective technique for limiting the rate at which clients can access an API, particularly to protect against Distributed Denial of Service (DDoS) attacks or abusive users. Throttling based on IP addresses ensures that individual clients, identified by their IP addresses, cannot overwhelm your API with too many requests in a short period.

### INFO:

*A Denial of Service (DoS) attack overwhelms a server or network with excessive requests, rendering it unavailable to legitimate users. By exhausting resources like CPU or bandwidth, the target becomes unresponsive. Distributed Denial of Service (DDoS) amplifies the attack using multiple sources. These attacks can cause significant downtime and damage. Defenses like throttling, caching, and IP-based rate limiting help protect against them.*

To apply IP-based throttling globally, we need to add **AnonRateThrottle** to the **DEFAULT\_THROTTLE\_CLASSES**.

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',  # For
        unauthenticated users (by IP)
    ]
    ...
}
```

## Conclusion

In this chapter, we have explored OpenAPI and tools to auto-generate, maintain, and present clear API documentation that is interactive and user-friendly. We also explored essential performance optimization techniques like caching and rate limiting, which enhance response times and safeguard your API. Implementing these strategies ensures that your API not only functions efficiently but is also secure and easy to use. With these practices in place, you can build APIs that scale effectively while offering a seamless experience to developers and users alike.

In the next chapter, we will explore the key steps to prepare a data science API for production, covering essential deployment strategies and best practices to ensure a smooth and efficient launch.

## Questions

1. Why is auto-generated documentation important in API development?
2. What is caching, and how does it help improve API performance?

3. What is rate limiting, and how does it protect your API from overuse or abuse?
4. What is the role of OpenAPI documentation in ensuring an API acts as a contract between developers and users?
5. Why is it important to document both GET and POST methods explicitly in an API?
6. How does rate limiting protect an API from overuse, and what are some common rate limiting strategies in Django Rest Framework?
7. How can IP-based throttling protect an API from Distributed Denial of Service (DDoS) attacks, and how is it implemented in Django Rest Framework?

## **Exercises**

1. Create an endpoint in Django Rest Framework, document it using the **@extend\_schema** decorator, and ensure that it displays parameters, request body, and response formats in the generated OpenAPI documentation.
2. Implement caching for a Django API endpoint using **@cache\_page** and verify that the response is cached for 15 minutes.

[OceanofPDF.com](https://oceanofpdf.com)

## **CHAPTER 8**

# **Deploying Your Data Science API**

## **Introduction**

In this chapter, we will explore the process of deploying a Django-based data science API using industry-standard tools and best practices. You will learn how to configure Gunicorn as a production-grade WSGI server and build optimized Docker images for seamless deployment.

We will also dive into Kubernetes, the leading container orchestration platform, to manage your API at scale. This includes setting up a Kubernetes cluster, deploying your containerized API, and using Kubernetes features like ConfigMaps, Secrets, StatefulSets, and Probes to ensure your application runs efficiently and reliably. By the end of this chapter, you will have a clear understanding of how to deploy your API in a production environment, ready to handle real-world workloads.

## **Structure**

In this chapter, we will cover the following topics:

- Introduction to Gunicorn
- Configuring Gunicorn for Django deployment
- Understanding and Creating Dockerfiles for Django
- Using the Image Registry
- Introduction to Kubernetes
- Configuring a Kubernetes Cluster for a Django Application
- Adding Liveness and Readiness Probes

## **Introduction to Gunicorn**

Python Web Server Gateway Interface (WSGI) is a specification that describes how a web server communicates with Python web applications.

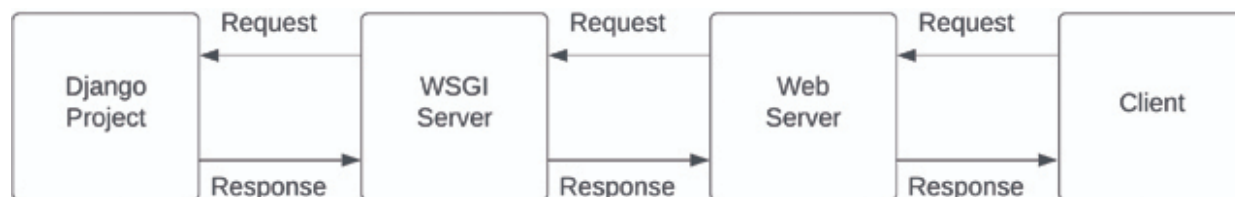
The standard is described in PEP 3333 (<https://peps.python.org/pep-3333/>). Gunicorn is a WSGI HTTP server for UNIX systems. It is compatible with many web frameworks, such as Django and Flask. Gunicorn is commonly used in production environments and is known for its efficiency and speed.

Regarding architecture, WSGI acts as an intermediary interface between the Django application and the web server.

#### INFO:

*A reverse proxy is a server that sits in front of web servers and forwards client requests to those web servers. A reverse proxy protects servers by hiding their identities from clients. When a client requests a resource, the reverse proxy is the “face” of the back-end servers.*

The request originates from the client or browser and is sent to the web server. The web server forwards this request to the appropriate WSGI server and finally reaches the Django project. The response does the same but in the backward direction.



*Figure 8.1: How Gunicorn works*

Gunicorn forks multiple worker processes to handle requests. Each worker is a separate process with its own memory space and instance of the Python application. It supports synchronous workers based on traditional multi-threaded or multi-process web servers and asynchronous workers.

## Configuring Gunicorn for Django Deployment

In this section, we will configure our Django project for using Gunicorn in production. This configuration requires some changes in our settings to guarantee a safe environment and the installation of Gunicorn.

Our project's `settings.py` file, located in `src/recommendation_system/settings.py`, contains development settings like `DEBUG = True`, which are unsuitable for a production environment. We



can use the environment variable **DJANGO\_SETTINGS\_MODULE** to specify which configuration to use in production. Using this environment variable, we will create a shared settings file called **src/recommendation\_system/base.py** and two more files, including **src/recommendation\_system/dev.py** and **src/recommendation\_system/production.py**.

Both **dev.py** and **production.py** will inherit shared configuration from **base.py** and the other two files will contain the corresponding values for each environment.

Rename the file from **src/recommendation\_system/settings.py** to **src/recommendation\_system/base.py**. Including the **base.py** in the book will be too extensive, you can check the **chapter\_8 branch** of the GitHub repository <https://github.com/ava-orange-education/Ultimate-Web-API-Development-with-Django-REST-Framework> to review its full content.

Now let us create the **development** **src/recommendation\_system/dev.py** file with the following contents:

```
from .base import * # noqa
DEBUG = True
ALLOWED_HOSTS = ["localhost", "127.0.0.1"]
SECRET_KEY = os.getenv("SECRET_KEY", "development_env")
CORS_ALLOWED_ORIGINS = [
    "http://localhost",
    "http://127.0.0.1",
]
```

As you can see, we are adding configurations only related to the development environment that we do not want to have enabled in production. Let us review the **dev.py** settings:

- **DEBUG** is set to **True**. Having more verbosity when an error occurs is beneficial for the development environment. Using **DEBUG** will help troubleshoot problems while coding.
- **ALLOWED\_HOSTS** is set to **['127.0.0.1', 'localhost']** to restrict access to the web application, ensuring it only responds to requests from these trusted local addresses during development.
- **CORS\_ALLOWED\_ORIGINS** is set to **['127.0.0.1', 'localhost']** to allow the web application to accept requests from these trusted origins during

development, ensuring that local resources can interact without cross-origin restrictions.

For production, we have the following settings in the `src/recommendation_system/production.py` file:

```
from .base import *

DEBUG = False
ALLOWED_HOSTS = os.getenv("ALLOWED_HOSTS",
                           "localhost").split(",")
SECRET_KEY = os.getenv("SECRET_KEY")
```

Let us review each of the settings:

- **DEBUG:** It is set to **False**. These are the recommended settings for production. Having it enabled could lead to information leaks and security incidents.
- **ALLOWED\_HOSTS:** This configuration ensures that Django will only allow requests sent to `yourproductiondomain.com` and reject requests to any other domain.

Since we made a refactoring, we must change all the references to `recommendation_system.settings` to the new `recommendation_system.dev` and `recommendation_system.production`.

Open these files and change them accordingly:

- **src/manage.py:** Change the default value of **DJANGO\_SETTINGS\_MODULE** to `recommendation_system.dev` in the line:  
`os.environ.setdefault("DJANGO_SETTINGS_MODULE", ...)`
- **src/mypy.ini:** Change **DJANGO\_SETTINGS\_MODULE** to `recommendation_system.dev`.
- **src/recommendation\_system/asgi.py:** Change the default value to `recommendation_system.production` in the line:  
`os.environ.setdefault("DJANGO_SETTINGS_MODULE", ...)`
- **src/recommendation\_system/wsgi.py:** Change the default value to `recommendation_system.production` in the line:  
`os.environ.setdefault("DJANGO_SETTINGS_MODULE", ...)`

- **src/recommendation\_system/celery.py**: Change the default value to **recommendation\_system.production** in the line:

```
os.environ.setdefault("DJANGO_SETTINGS_MODULE", ...
```

Now, with our settings refactor, we are ready to install Gunicorn using rye:

```
rye add gunicorn
rye sync
```

Before configuring Gunicorn for production, we need to test it with our project to ensure it works as expected:

```
export SECRET_KEY=test
rye run gunicorn --workers 4
recommendation_system.wsgi:application
```

By default, Gunicorn listens on port 8000. Open on your browser the URL <http://127.0.0.1:8000> and verify that the login page is shown. The **--workers** parameter specifies the number of worker processes to handle the requests.

Gunicorn can be configured by using a configuration file. Since we want to deploy the project to Kubernetes, we will extend this file with more settings. Let us start with some basic settings. Create a new file in the **src** directory at the same level as the **manager.py** file. Let us call it **gunicorn.conf.py**.

```
bind = "0.0.0.0:8000"
workers = 3
accesslog = "-"
errorlog = "-"
```

Binding to 0.0.0.0 means that Gunicorn can accept requests coming from any IP address that can reach the server. Running inside a Docker container on Kubernetes typically takes requests from any source.

The choice of 3 workers is a starting point. Determining the optimal number of workers depends on various factors, which will be discussed in the subsequent sections.

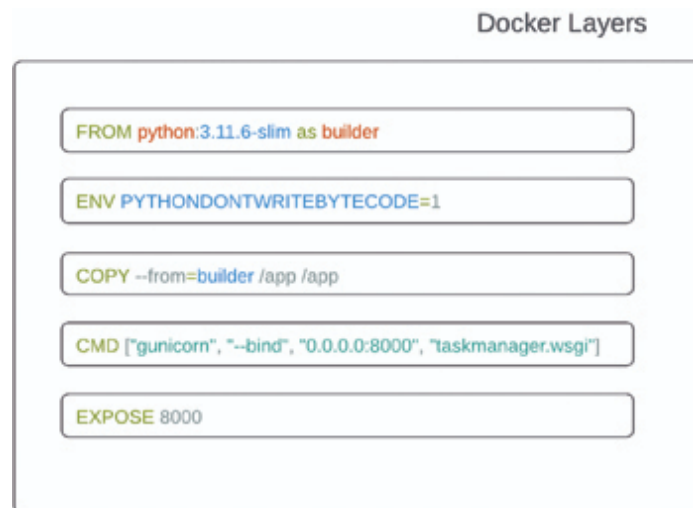
These settings configure where Gunicorn will write its access and error logs. The “-” setting tells Gunicorn to output the logs to stdout and stderr, respectively. These settings are typical because Docker captures anything written to stdout and stderr.

## Understanding and Creating Dockerfiles for Django

Containers are lightweight packages that contain everything needed to run a software application, including code, runtime, libraries and settings. Containers will run the software in a reproducible environment in different computing environments.

We already configured Docker to use it in a local environment. Now, we will use Docker again, but for our production environment. To deploy our project into production, we will need a container image. To create a container image, we need to write a Dockerfile file. A Dockerfile contains instructions where most of them create a new layer, while others may only alter metadata. Docker images are made up of layers; each layer is read-only except for the last one.

Following is an example of a Dockerfile with 5 instructions, resulting in 5 layers:



*Figure 8.2: Representation of Docker layers*

Here is a list of some useful Dockerfile instructions:

- **FROM:** Creates a base layer from an image.
- **ENV:** Sets an environment variable and creates a layer with this configuration.
- **RUN:** Executes commands and any files created during the process form a new layer.

- **COPY:** Copies files from the local file system into the container, creating a layer with these files.
- **WORKDIR:** Sets the current working directory for any subsequent Dockerfile instructions.
- **EXPOSE:** Informs Docker that the container listens to specific network ports at runtime.
- **CMD:** Provides the default command and arguments for an executing container.

Layers are stacked on top of each other. When you run a container, Docker takes all these read-only layers and adds a read-write layer. Any changes you make to the container file system, such as writing new files, modifying existing files, or deleting files, are made in this writable layer.

#### **INFO:**

##### Best Practices with Layers-

- ***Minimize the Number of Layers:*** Combine related commands into a single RUN instruction where it makes sense to reduce the number of layers.
- ***Clean up Within Layers:*** For example, if you install packages with apt-get, you should clean the cache within the same RUN command to prevent the cache from becoming a permanent part of the layer.
- ***Use .dockerignore:*** To avoid adding unnecessary files to your Docker context, which can unnecessarily increase the size of the built images.

Docker has a feature called multi-stage. Multi-stage allows you to create smaller images that are cleaner and more secure. The Dockerfile will have more than one stage and each stage can copy artifacts from these intermediate stages.

Our Dockerfile will comprise two stages: the build and the runtime stage. The build stage will create the virtual environment and the image will have all required dependencies. The build stage will also collect the static files.

The runtime stage will copy the virtual environment from the build stage and prepare the Gunicorn command.

Let us start with the build stage. At the root of the project directory, create a new file at the **src** directory called Dockerfile and populate it with the following content:

```
# --- Build Stage ---
FROM python:3.12.2-slim AS builder

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PATH="/root/.rye/bin:$PATH" \
    UV_LINK_MODE=copy \
    UV_COMPILE_BYTECODE=1 \
    UV_PYTHON_DOWNLOADS=never \
    UV_PYTHON=python3.12 \
    UV_PROJECT_ENVIRONMENT=/app \
    DJANGO_SETTINGS_MODULE=recommendation_system.production

# Install dependencies
RUN apt-get update \
    && apt-get install -y --no-install-recommends build-essential \
    curl libpq-dev

# Install uv
COPY --from=ghcr.io/astral-sh/uv:latest /uv /usr/local/bin/uv

# Copy the project files into the builder stage
RUN uv venv /app

WORKDIR /app

COPY pyproject.toml /app
COPY requirements.lock /app/uv.lock
COPY README.md /app
COPY src/manage.py /app

# Install project dependencies with uv
RUN uv sync

# Copy the rest of the application's code
COPY src/ /app/
```

The first stage of this **Dockerfile** defines a **builder** stage, which is responsible for preparing the application environment, installing

dependencies, and ensuring the code is ready for production. Here is a step-by-step explanation of each part of the build stage:

```
FROM python:3.12.2-slim AS builder
```

This line sets the base image to **python:3.12.2-slim**. The slim version of the image is a lightweight version, which reduces the overall image size by excluding unnecessary components. The specific Python version 3.12.2 ensures consistency, making the build process more reproducible and stable across environments. The stage is named **builder**, which allows it to be referenced later during multi-stage builds.

```
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    UV_LINK_MODE=copy \
    UV_COMPILE_BYTECODE=1 \
    UV_PYTHON_DOWNLOADS=never \
    UV_PYTHON=python3.12 \
    UV_PROJECT_ENVIRONMENT=/app \
    DJANGO_SETTINGS_MODULE=recommendation_system.production
```

Setting environment variables is critical for configuring how the application runs inside the container. These are explained as follows:

- **PYTHONDONTWRITEBYTECODE=1**: Prevents Python from generating **.pyc** files, which are bytecode-compiled versions of the Python files. These are not necessary for the Docker image and would waste space.
- **PYTHONUNBUFFERED=1**: Ensures that Python outputs are unbuffered, meaning logs and other outputs appear immediately in the console, which is useful for real-time logging.
- **PATH="/root/.rye/bin:\$PATH"**: Updates the system's executable search path to include the directory where Rye (a Python package and environment manager) is installed. This ensures that commands like **uv** (Rye's universal command-line tool) can be accessed anywhere within the container.
- **UV\_LINK\_MODE=copy**, **UV\_COMPILE\_BYTECODE=1**,  
**UV\_PYTHON\_DOWNLOADS=never**, **UV\_PYTHON=python3.12**,  
**UV\_PROJECT\_ENVIRONMENT=/app**: These variables configure Rye and **uv** to control the Python environment and dependencies. They instruct Rye

to work with Python 3.12, avoid downloading Python versions, and set the project environment to /app.

- **DJANGO\_SETTINGS\_MODULE=recommendation\_system.production:** This sets the Django configuration to use the production settings, ensuring that when the container runs, it operates in production mode (rather than development or testing).

```
RUN apt-get update \
    && apt-get install -y --no-install-recommends build-
    essential curl libpq-dev
```

This command installs essential system dependencies that are needed to compile Python packages and interact with PostgreSQL databases:

- **build-essential:** A package that includes the compiler and other build tools needed to install Python packages with C extensions.
- **curl:** A tool for downloading files from the internet, which will be used to install additional dependencies later.
- **libpq-dev:** The PostgreSQL development headers, which are required for working with PostgreSQL databases via Python libraries like `psycopg2`.

Using the **--no-install-recommends** flag ensures that only essential packages are installed, keeping the image lightweight by avoiding unnecessary dependencies.

```
COPY --from=ghcr.io/astral-sh/uv:latest /uv /usr/local/bin/uv
```

This line copies the **uv** binary (which comes from the Rye toolset) from a pre-built image hosted at `ghcr.io`. The **COPY** command pulls the binary from the latest version of **uv** and places it in **/usr/local/bin**, making it accessible system-wide. This step ensures that the Rye tool is available for managing Python dependencies and environments.

```
RUN uv venv /app
```

This command creates a virtual environment for the Python application inside the **/app** directory using Rye's **uv** tool. A virtual environment isolates the project's dependencies from the system Python installation, ensuring consistency and avoiding conflicts.

```
WORKDIR /app
```



This command sets the current working directory for all subsequent commands to **/app**. Any files or commands executed after this will operate within the **/app** directory. This is where the Python application code will live.

```
COPY pyproject.toml /app
COPY requirements.lock /app/uv.lock
COPY README.md /app
COPY src/manage.py /app
```

These **COPY** commands transfer specific project files into the container. These files are necessary for installing dependencies and setting up the application:

- **pyproject.toml**: Defines the project's dependencies and build system (for use with Rye).
- **requirements.lock** → **/app/uv.lock**: Contains the locked dependencies required for the project. Renaming it to **uv.lock** ensures compatibility with Rye's conventions.
- **README.md**: The project documentation, which might be used during the build or for future reference.
- **manage.py**: The entry point for managing the Django application.

```
RUN uv sync
```

This command installs the Python dependencies as specified in the **uv.lock** (formerly **requirements.lock**) file. It uses Rye's **sync** command to ensure that all necessary dependencies are installed inside the virtual environment, exactly matching the locked versions to ensure consistency across environments.

```
COPY src/ /app/
```

Finally, this command copies all the application source code from the local **src/** directory into the container's **/app/** directory. Copying the code at this stage ensures that the dependencies are installed first (which allows Docker to cache that step if the dependencies have not changed), minimizing rebuild times when the code is updated.

This build stage prepares everything needed to run the application in a production environment. It sets up the Python environment, installs all required dependencies, and ensures the application code is available.

Our build stage is ready. Our next step is to create the production runtime stage. Just after the build stage, append the following instruction to the Dockerfile:

```
# --- Production Stage ---
# Define the base image for the production stage
FROM python:3.12.2-slim AS production

RUN apt-get update \
    && apt-get install -y --no-install-recommends libpq-dev

# Copy virtual env and other necessary files from builder stage
# Copy installed packages and binaries from builder stage
COPY --from=builder /app /app

# Set the working directory in the container
WORKDIR /app

# Set user to use when running the image
# UID 1000 is often the default user
RUN groupadd -r django && useradd -r -g django -d /app -s \
    /bin/bash django && \
    chown -R django:django /app

USER django

# Start Gunicorn with a configuration file
CMD ["/app/bin/gunicorn", "--bind", "0.0.0.0:8000",
    "recommendation_system.wsgi"]

# Inform Docker that the container listens on the specified
network ports at runtime
EXPOSE 8000
```

You can test the new Docker image by running the following commands:

```
docker build -t my-app .
docker run -d -p 8000:8000 my-app
```

You can check the logs by executing:

```
docker ps
# find the container id
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	

```
6ff5ff4165d0    my-app          "/app/bin/gunicorn -..."    6 days
ago    Up 6 days    0.0.0.0:8000->8000/tcp          laughing_hawking
```

Then you can check the logs with:

```
$ docker logs
6ff5ff4165d0459b627ca459ee034a82904ad6e472724643d534b7d09bac613
5
[2024-10-04 09:44:03 +0000] [1] [INFO] Starting gunicorn 23.0.0
[2024-10-04 09:44:03 +0000] [1] [INFO] Listening at:
http://0.0.0.0:8000 (1)
[2024-10-04 09:44:03 +0000] [1] [INFO] Using worker: sync
[2024-10-04 09:44:03 +0000] [6] [INFO] Booting worker with pid:
6
[2024-10-04 09:44:03 +0000] [7] [INFO] Booting worker with pid:
7
[2024-10-04 09:44:03 +0000] [8] [INFO] Booting worker with pid:
8
```

Open the URL <http://localhost:8000/api/docs> and ensure that you see the API documentation showing.

## Using the Image Registry

A container registry allows you to store the container images with version control. Registries are also a way to distribute your container and a critical step in deploying a container application. When working with Kubernetes, the registry will be used to download the images in the deployment or the applications, as seen in the following section.

When working with a registry, Docker provides commands to interact with it. You can download an image from the registry using the `docker pull` command.

```
docker pull ubuntu
```

There exist several container registries in the industry, and cloud providers usually offer a registry as a service. There exist public and private registries. The most popular public registry is Docker Hub, the default registry to pull images when using Docker.

For uploading the image to the registry, Docker provides the command `push`. `Push` will upload the image to the repository:

```
docker buildx create --use
docker buildx build --platform linux/amd64,linux/arm64 -t
llazzaro/django_rest_book:latest .
docker push llazzaro/django_rest_book:latest
```

For our example, since our GitHub repository is public, we will use the Docker Hub public repository to push and pull the image. However, for a production deployment, you must use a private repository. Exposing the Docker image to the public will allow anyone to access your application code, which will be a security incident in most scenarios. Make sure your repository permissions are correctly configured.

Once our image has been uploaded to the registry, we can pull it with the Docker command pull:

```
docker pull llazzaro/django_rest_book
```

## **Introduction to Kubernetes**

A container orchestration system is software that automates container deployment, management, scaling and networking. Kubernetes (K8s) is an open-source container orchestration system and probably the most famous one. Kubernetes groups containers that make an application into logical units for easy management and discovery.

Before deploying our application to Kubernetes, we must cover some crucial components.

For deploying our application, it is not necessary to know how Kubernetes works internally. However, some knowledge will help us troubleshoot and understand how our application runs.

## **Cluster**

The Kubernetes cluster is composed of nodes. All these nodes allow you to scale your application across different machines in terms of nodes. The cluster also has self-healing properties. When something fails, the scheduler will try to restart the failed container and reschedule it.

## **Node**

Nodes are workers that run container applications. A node is a single machine in the Kubernetes cluster. This machine could be physical or virtual. There are two types of nodes: the master node and the worker node. The master node is responsible for scheduling and responding to cluster events. The other type of node is the worker node. These machines run your applications and are managed by the master node.

Each node has several components, but we just abstract everything as a node to keep it simple. The Kubernetes documentation (<https://kubernetes.io/docs/home/>) is an excellent source for learning more about node components.

## **Scheduler**

The Kubernetes scheduler selects the most suitable node to run your application. The scheduler is a critical component and fundamental to workload management.

## **Pods**

A pod is a set of one or more containers that share common resources like storage, network and a specification on how to run the containers. Kubernetes orchestrates containerized applications with a dynamic lifecycle, managing pods as non-permanent entities. Containers within a pod may be terminated by the scheduler to manage system resources, respond to application scaling directives, perform updates, or recover from failures.

While containers in Kubernetes can use local filesystems and memory, it's important for developers to recognize that such storage is ephemeral. Any data saved locally can be lost when the container is restarted or moved by the scheduler, therefore, for persistent storage, they should utilize Kubernetes volumes or external storage systems. While pods are commonly used for stateless services, they can also be part of stateful applications, particularly when managed through Kubernetes StatefulSet objects.

## **Deployments**

Deployment is a way to specify to Kubernetes how to create or modify instances of the pods. Deployments are ideal for stateless applications and

help roll out updates, roll back versions and scaling applications.

## ReplicaSets

A ReplicaSet in Kubernetes is a mechanism that ensures a specific number of identical pod replicas are always running. It automatically replaces Pods that fail, get deleted or are terminated. ReplicaSets are crucial for ensuring high availability and resilience of applications.

## Services

Services provide a persistent endpoint to access the pods distributed across one or more nodes. There are different types of services:

- **ClusterIP:** A **ClusterIP** service in Kubernetes provides a stable internal IP address for accessing a set of pods from within the cluster.
- **NodePort:** A **NodePort** service exposes a service on the same port of each selected node in the cluster, making it accessible from outside the cluster using `<NodeIP>:<NodePort>`.
- **LoadBalancer:** A **LoadBalancer** service in Kubernetes automatically integrates with the cloud provider's load balancer, allowing external traffic to be evenly distributed to the pods.

## ConfigMaps and Secrets

A ConfigMap is a store for non-confidential data as key-value pairs. The store can be used to save configuration files, environment variables, and command line arguments.

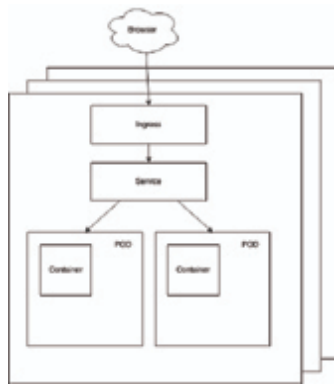
Secrets is similar to ConfigMap, but it provides an extra layer of security, and its purpose is storing secrets.

## Ingress

Ingress in Kubernetes is a key component for managing incoming traffic to services within a cluster. Acting as the entry point for external access, it routes external requests to the appropriate services. Ingress often includes capabilities like load balancing and SSL termination, effectively directing and securing communication with services inside the cluster.

## StatefulSets

The pod component is ideal for stateless applications. However, some applications require persistent storage or stable network identifiers. Typically, this is used for databases or caching services. In this chapter, we will use StatefulSets, but we recommend using managed services whenever possible. Most cloud providers offer managed database services that will reduce maintenance and management overhead. Managed services also provide robust security features and high reliability. Using StatefulSet is still possible, but your team or company must manage these essential and critical resources.



*Figure 8.3: Kubernetes architecture overview*

The Kubernetes architecture diagram represents the core components of a Kubernetes cluster. At the forefront is the Ingress, which acts as the entry point for external traffic, directing requests to the appropriate services. Service serves as an abstraction layer that efficiently manages access to the Pods. Pods are the smallest deployable units in Kubernetes. Each Pod encapsulates one or more containers, which are running instances of applications. This arrangement illustrates the flow from external access to internal process management.

## Configuring a Kubernetes Cluster for a Django Application

The most basic Kubernetes configuration requires a deployment and a service. Let us start with a deployment, create a new file in the new directory

**k8s** at the root of the repository, then create a new file in this directory with the name **deployment.yaml** and populate it with the following contents:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: recommendation-system-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: recommendation-system-app
  template:
    metadata:
      labels:
        app: recommendation-system-app
    spec:
      containers:
        - name: recommendation-system-app
          image: llazzaro/django_rest_book
          ports:
            - containerPort: 8000
```

Every Kubernetes configuration file needs to specify the API version. In this particular file, we have defined it to use **app/v1**. The kind indicates the type of resource we will configure, which in this case is deployment.

We use the metadata field to set a unique name. As a convention, we use the **postfix -deployment**. Using this name will be helpful when troubleshooting problems.

The spec (specification) section sets the replicas to 3. The replicas allow Kubernetes to have three instances of the pod for the deployment. This is a way to ensure availability and load distribution for our project.

Then we use the **select** to instruct Kubernetes which application the specification has to be applied, which is the **recommendation-system-app**.

Under the **template**, the spec for the containers within the Pods is defined. It specifies that each Pod should run a single container, named recommendation-system-app and that this container should use the Docker image **llazzaro/recommendation-system**.



In this configuration, we use **llazzaro/recommendation-system**, pulling the image from the Docker Hub public registry. You must change this to point to the appropriate registry when using a private registry.

The ports section under the container spec exposes port 8000 of the container, suggesting that the Django application listens on this port for incoming HTTP traffic. In this case, our port was set to the same one our Docker file exposes since it is the port where our Gunicorn is listening to serve the Django application.

Next, we create a new file for the service configuration in the **k8s** directory. Let us call it **services.yaml** and populate it with the following contents:

```
apiVersion: v1
kind: Service
metadata:
  name: recommendation-system-service
spec:
  type: NodePort
  selector:
    app: recommendation-system-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8000
```

The file starts with an **apiVersion** set to v1 and metadata to form the unique name **recommendation-system-service** using the **postfix -service**. This name will be handy for identifying the resource in the Kubernetes cluster for troubleshooting.

This file creates a service using NodePort. The selector uses the same application name, **recommendation-system-app**, to apply this service specification. NodePort will expose the service on each cluster node's IP and use the static port 80 protocol TCP that HTTP uses. The **targetPort** settings indicated to which port in the Pod the service will forward the traffic.

With these two files, we are ready to try our first deployment. There are several ways to configure the Kubernetes cluster. We will use the official **kubect1** command.

<b>NOTE:</b>
--------------

To install Kubectl, follow the official installation instructions on the Kubernetes website: <https://kubernetes.io/docs/tasks/tools/>.

Installing kubectl should be straightforward.

You can also have a local Kubernetes cluster using minikube or the official Docker.

We recommend following the official minikube installation at the website: <https://minikube.sigs.k8s.io/docs/start/>

If you are willing to use official Docker, you can follow these steps: <https://www.docker.com/products/kubernetes/>

Our first step will be to create a new Kubernetes namespace. A Kubernetes namespace is a way to divide a cluster of resources between multiple applications logically.

```
kubectl create namespace recommendation-system
>namespace/recommendation-system created
```

To apply the deploy and service config to the cluster, we need to execute two commands:

```
kubectl apply -f k8s/deployment.yaml --
namespace=recommendation-system
deployment.apps/recommendation-system-deployment created
kubectl apply -f k8s/services.yaml --namespace=recommendation-
system
service/recommendation-system-service created
```

To see the pods created by the deployment, let us execute the following command:

```
$ kubectl --namespace recommendation-system get pods -l
app=recommendation-system-app
```

NAME	READY	STATUS	RESTARTS	AGE
recommendation-system-deployment-667bd87b6-gxv7c	1/1	Running	0	31s
recommendation-system-deployment-667bd87b6-lsgch	1/1	Running	0	31s
recommendation-system-deployment-667bd87b6-mgjs9	1/1	Running	0	31s

You can check the logs of each container with the command:

```
$ kubectl --namespace recommendation-system logs
recommendation-system-deployment-667bd87b6-gxv7c
[2023-11-14 18:46:38 +0000] [1] [INFO] Starting gunicorn 21.2.0
[2023-11-14 18:46:38 +0000] [1] [INFO] Listening at:
http://0.0.0.0:8000 (1)
[2023-11-14 18:46:38 +0000] [1] [INFO] Using worker: sync
[2023-11-14 18:46:38 +0000] [7] [INFO] Booting worker with pid:
7
[2023-11-14 18:46:38 +0000] [8] [INFO] Booting worker with pid:
8
[2023-11-14 18:46:38 +0000] [9] [INFO] Booting worker with pid:
9
```

Now you can access your service using the node IP address and with the exposed service port. You can get the service port with the **kubectl** command:

```
$ kubectl get services --namespace recommendation-system
NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP
IP  PORT(S)          AGE
recommendation-system-
service  NodePort  10.107.151.88  <none>        80:30893/TCP
8m52s
```

In my environment, the node port is 30893, but Kubernetes can assign any number in the range of 30000-32767 by default. You can get the node IP address with the command:

```
kubectl get nodes -o wide
```

If you are using macOS and Docker desktop, you can use localhost and the node port.

We still need to set the environment variables for a proper configuration of our Django application. In the same k8s directory, let us create a new file with ConfigMaps called **configs.yaml** with the following content:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: recommendation-system-settings
```

```
data:
  DEBUG: "False"
  ALLOWED_HOSTS: "localhost"
  DB_NAME: "mydatabase"
  DB_USER: "postgres"
  DB_HOST: "postgres"
  DB_PORT: "5432"
```

In this configuration file, we define a ConfigMap with the same recommendation-system settings. In the data section, we set the values of the environment variables for our containers. **DEBUG** mode is set to **False**, which indicates a production environment. The **ALLOWED\_HOSTS** is configured for “localhost,” in a natural production environment. This setting should be set to your domain. Database configurations such as **DB\_NAME**, **DB\_USER**, **DB\_HOST**, and **DB\_PORT** set the values to connect to a PostgreSQL database named “mydatabase” running on localhost with the default port “5432.”

Let us apply the ConfigMap to the cluster:

```
kubectl --namespace recommendation-system apply -f
k8s/configs.yaml
```

Using StatefulSet for a PostgreSQL database is a common approach to deploying an application that requires persistent storage and stable network identity. Using a StatefulSet for a production environment has several drawbacks and will increase your projects’ maintenance costs. For practical reasons, we will use StatefulSets in this chapter. However, always consider using managed cloud solutions.

The first step is to set up a persistence storage for PostgreSQL. We need to understand two concepts when setting up persistent volumes (PV).

- **PersistentVolume:** Storage in the cluster that has been provided by an administrator or dynamically provided using a storage class
- **PersistentVolumeClaim:** This is a way to request the usage of the storage by a project. This is usually specified in the StatefulSet configuration file.

You don’t have to manually create a persistent volume when using the Docker desktop for Kubernetes. The storage class will automatically create a persistent volume that meets the requirements of the volume claim. If you

use StatefulSets in Kubernetes, you may need to request the persistent volume to your cluster administrator.

Let us create a new file in the **k8s** directory called **statefulset-postgresql.yaml** with the following contents:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  serviceName: "postgres"
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:16
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_DB
              valueFrom:
                configMapKeyRef:
                  name: recommendation-system-settings
                  key: DB_NAME
            - name: POSTGRES_USER
              valueFrom:
                configMapKeyRef:
                  name: recommendation-syste-settings
                  key: DB_USER
            - name: POSTGRES_PASSWORD
              valueFrom:
```

```

      secretKeyRef:
        name: recommendation-system-secrets
        key: DB_PASSWORD
    - name: PGDATA
      value: /var/lib/postgresql/data/pgdata
  volumeMounts:
    - name: postgres-storage
      mountPath: /var/lib/postgresql/data
  volumeClaimTemplates:
    - metadata:
        name: postgres-storage
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 10Gi

```

The StatefulSet name is PostgreSQL and uses a single replica. The container uses the PostgreSQL image 16. Configuration of the environment variables is defined in the previous **ConfigMap**, and the **POSTGRES\_PASSWORD** will be defined in the Kubernetes Secret to ensure it is secured appropriately. A volume mount (**/var/lib/postgresql/data**) is defined for data persistence, which is linked to a PersistentVolumeClaim named **postgres-storage**. This claim requests 10Gi of storage and will be dynamically provisioned with the **ReadWriteOnce** access mode, indicating that a single node can mount the volume as read-write. The other PostgreSQL settings use the **ConfigMap** values from the recommendation-system-settings.

Let's apply the new statefulset to the cluster:

```

kubectl --namespace recommendation-system apply -f
k8s/statefulset-postgresql.yaml

```

Verify its status with the get StatefulSet command of **kubectl**:

```

$ kubectl get statefulset --namespace recommendation-system
NAME      READY   AGE
postgres  1/1     63s

```

We still need to set the **DB\_PASSWORD**, **SECRET\_KEY** and **JWT** secrets, since those are secret values, we need to store them in the secrets API of Kubernetes.

The PostgreSQL StatefulSet requires a service:

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  type: ClusterIP
  ports:
    - port: 5432
      targetPort: 5432
      protocol: TCP
  selector:
    app: postgres
```

Apply the new config:

```
kubectl --namespace recommendation-system apply -f
k8s/postgresql-service.yaml
```

The next step is to create the file with the secrets, but since this file will contain secrets in plain, we need to encrypt this file before committing it to the repository.

The `sops` (Secrets OPerations) is an open-source tool (<https://github.com/getsops/sops>) used to encrypt, decrypt, and edit sensitive data files. Sops support key management services like AWS Key Management Service (KMS), Google Cloud KMS, Azure Key Vault, and PGP. Sops is format agnostic and it could be used for JSON, YAML, and other types of files.

**INFO:**

*You can install **sops** in macOS with **brew**:*

```
brew install sops
```

*in linux:*

```
apt install sops
```

We are going to use sops with PGP, the first step after installing PGP is to generate a key:

```
gpg --full-generate-key
```

After generating your key, list your GnuPG keys to get your key ID.

```
gpg --list-secret-keys --keyid-format LONG
```

Save the **pgp-key-id** of the newly generated key since we are going to use it for encrypting the secrets file we are going to create in the `k8s` directory with the following contents a file named **secrets.yaml**:

```
apiVersion: v1
kind: Secret
metadata:
  name: recommendation-system-secrets
type: Opaque
stringData:
  SECRET_KEY: "your-secret-key"
  DB_PASSWORD: "your-db-password"
```

This configuration file creates a secret with the name `recommendation-system-secrets` with the secrets **SECRET\_KEY** and **DB\_PASSWORD**.

```
GPG_TTY=$(tty)
export GPG_TTY
sops --pgp <pgp-key-id> -e -i k8s/secrets.yaml
```

Sops will open a text editor with the decrypted contents. Once you quit the editor, it will encrypt the file if you try to open the file **k8s/secrets.yaml** directly with a text editor. You see that the values of the **DB\_PASSWORD** and **SECRET\_KEY** are encrypted.

Using sops will allow us to commit the file to the repository safely.

#### **INFO:**

*When using sops, it is required to decrypt the file to apply it to the Kubernetes cluster. If the pgp secret is only owned by one team member, that could be a problem since it will be the only one with this power. When using a cloud key manager, you can share the key among team members or even with a CI/CD pipeline.*

To apply the secrets to the cluster, we first need to decrypt it and apply the decrypted secrets. You can do it with the following command:

```
sops -d k8s/secrets.yaml | kubectl apply -f -
```



Having the ConfigMaps and Secrets is not enough to have the environment variables set in our containers, we need to change our **deployment.yaml** file to instruct this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: recommendation-system-deployment
spec:
  replicas: 1
  ...
  template:
    ...
    spec:
      containers:
      - name: recommendation-system-app
      ...
      envFrom:
      - configMapRef:
          name: recommendation-system-settings
      - secretRef:
          name: recommendation-system-secrets
```

With the new change in the deployment configuration file, we describe how to set the environment variables using a **ConfigMap** and secrets referenced by the name.

Re-apply the deployment configuration to the cluster:

```
kubectl apply -f k8s/deployment.yaml
```

**NOTE:**

*If you need to force a new deployment to refresh secrets or config, you can rollout and restart the deployment with the command:*

```
kubectl rollout restart deployment recommendation-system-
deployment
```

For troubleshooting, it is helpful to tail the logs of all the containers by application label, the command is:

```
kubectl get pods -l app=recommendation-system-app -o name |  
xargs -I {} kubectl logs --tail=10 -f {}
```

Our application is almost ready to be used. However, we need a way to run the migrations. There are many alternatives to execute the migrations. We will opt to create a Kubernetes job that will use the **ConfigMaps** and secrets to perform the migration.

```
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: recommendation-system-migrate  
spec:  
  template:  
    spec:  
      containers:  
        - name: recommendation-system-app  
          image: llazzaro/web_applications_django  
          command: ["python", "manage.py", "migrate"]  
          env:  
            - name: DJANGO_SETTINGS_MODULE  
              value: "recommendation_system.production"  
          envFrom:  
            - configMapRef:  
                name: recommendation-system-settings  
            - secretRef:  
                name: recommendation-system-secrets  
      restartPolicy: Never  
      backoffLimit: 4
```

This Kubernetes configuration file defines a job resource named **recommendation-system-migrate**, which will execute database migrations for our project. It specifies a container named **recommendation-system-app** using the image **llazzaro/django\_rest\_book**. The primary command executed in this container is **python manage.py migrate**, which applies database migrations as defined in the Django project. The environment variable **DJANGO\_SETTINGS\_MODULE** is set to **recommendation\_system.production**, ensuring the correct Django settings are used. The **restartPolicy** is set to **Never**, indicating that the job should not be restarted automatically if it fails or completes. The **backoffLimit** is

set to 4, which limits the number of retries for the job to four attempts in case of failure. This job resource effectively automates the process of database migration.

Now apply the new jobs configuration:

```
kubectl --namespace recommendation-system apply -f
k8s/migration-job.yaml
```

Once the configuration is applied, Kubernetes will execute it and our database will be migrated.

You can check the status of the execution when getting the pods:

```
$ kubectl get po --namespace recommendation-system
NAME                                READY
STATUS  RESTARTS   AGE
postgres-0                                1/1    Running
0                4h33m
recommendation-system-deployment-8569bd7646-nvhgg
1/1  Running  0          16m
recommendation-system-migrate-hvm8t              0/1
Completed    0          17m
```

The status should be Completed. If you want to re-execute the migrations again, you will need to replace the current migration job with force:

```
kubectl replace --force -f k8s/migration-job.yaml --namespace
recommendation-system
```

If you now browse to the **nodeIP:nodePort** you should see the login page.

If you want to create a superuser using the management command, we can create an interactive bash shell with the command:

```
kubectl exec -it recommendation-system-deployment-74869f9d76-
9n7qt --namespace recommendation-system -- bash
django@recommendation-system-deployment-74869f9d76-9n7qt:/app$
python manage.py createsuperuser
```

The last step to finish the cluster configuration is to deploy a redis service using a StatefulSet. For this, we need to create the StatefulSet and the service.

Create a new configuration file in the **k8s** directory called **redis-statefulset.yaml** and populate it with the following configuration:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:7.0
          ports:
            - containerPort: 6379
          volumeMounts:
            - name: redis-data
              mountPath: /data
      volumeClaimTemplates:
        - metadata:
            name: redis-data
          spec:
            accessModes: ["ReadWriteOnce"]
            resources:
              requests:
                storage: 1Gi
```

This configuration file defines a **StatefulSet** named **redis**, which is used to deploy a Redis server within the cluster. The StatefulSet ensures stable and unique network identifiers and persistent storage for the Redis instance. It specifies a single replica, meaning one instance of the Redis server will be running.

The **selector** and **template** sections define the label **app: redis**, which is used to match the created pods with the StatefulSet. The **redis** container uses the image **redis:7.0** and opens port **6379**, the default port for Redis.

The **volumeMounts** section attaches a volume for persistent data storage to the **/data** directory inside the container, ensuring that Redis data persists across pod restarts.

The **volumeClaimTemplates** provide a template for creating a **PersistentVolumeClaim** named **redis-data**. This claim requests a storage volume of **1Gi** with **ReadWriteOnce** access mode, meaning the volume can be mounted as read-write by a single node. This volume is used by the Redis container to store data persistently.

We now have a single-node Redis instance in the Kubernetes cluster, with persistent storage to maintain data across restarts and deployments.

For the service to provide access to our redis instance, create a new file in **k8s/redis-service.yaml** with the following contents:

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  type: ClusterIP
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    app: redis
```

This configuration provides a stable network endpoint to access the Redis instance. It defines a **ClusterIP** service named **redis**.

The service exposes port **6379**, both as its port and **targetPort**. The selector with **app: redis** ensures this service routes network traffic to the correct pods, specifically those labeled as part of the Redis application.

By using a ClusterIP service, this configuration ensures that the Redis instance is consistently reachable at a known IP address within the cluster.

We will need to update the ConfigMap to use the new StatefulSet, edit the file **k8s/configs.yaml** and add the new environment variable with the

**redis** hostname:

```
REDIS_LOCATION: "redis://redis-0.redis.recommendation-  
system.svc.cluster.local/1"
```

Apply the new configuration to the cluster:

```
kubectl --namespace recommendation-system apply -f  
k8s/configs.yaml  
kubectl --namespace recommendation-system apply -f k8s/redis-  
service.yaml  
kubectl --namespace recommendation-system apply -f k8s/redis-  
statefulset.yaml  
# rollout restart the recommendation-system application  
kubectl rollout restart deployment recommendation-system-  
deployment --namespace recommendation-system
```

If you want to test if the redis connection works, you can log in and create a new task or open an interactive shell to connect to redis using the **kubectl exec** command.

## [Adding Liveness and Readiness Probes](#)

Probes are a way for Kubernetes to understand when the container is ready to start accepting traffic (readiness) and when it needs to be restarted (liveness).

Liveness probes are used to determine if the application within a container is running. When the liveness probe fails, Kubernetes will kill the container and restart it according to the restart policy. Be careful with adding external dependency checks like Redis or a database check to the liveness probes. If these services are temporarily unavailable, Kubernetes will keep restarting your containers.

Readiness determines if the application within the container is ready to service requests. When the readiness probe fails, Kubernetes will not send traffic to the pod, but it doesn't restart it. Some applications could take some time to load and this could be a useful use case.

We will implement the probes in a new Django application called Health. In this application, we will add two views, one for liveness and the other for readiness.

Create the new app with the **startapp** management command:

```
cd src
rye run python manage.py startapp health
```

Open the **src/recommendation-system/base.py** and add the health application:

```
INSTALLED_APPS = [
    # ...
    "health",
    # ...
]
```

In **health/views.py**, implement your health check logic:

```
from django.core.cache import cache
from django.db import connections
from django.http import JsonResponse

def liveness(request):
    # Perform checks to determine if the app is alive
    return JsonResponse({"status": "alive"})

def readiness(request):
    try:
        # Check database connectivity
        connections["default"].cursor()

        # Check cache (e.g., Redis) connectivity
        cache.set("health_check", "ok", timeout=10)
        if cache.get("health_check") != "ok":
            raise ValueError("Failed to communicate with cache backend")
        return JsonResponse({"status": "healthy"}, status=200)
    except Exception as e:
        return JsonResponse({"status": "unhealthy", "error": str(e)}, status=500)
```

The liveness probe returns a simple Json, while the readiness check verifies database and cache (Redis) connectivity in this example. If any check fails, it returns an unhealthy status.

In **health/urls.py**:

```

from django.urls import path
from . import views

urlpatterns = [
    path("liveness/", views.liveness_check,
         name="liveness_check"),
    path("readiness/", views.readiness_check,
         name="readiness_check"),
]

```

In the projects **src/recommendation-system/urls.py**, add the health application URLs:

```

...
urlpatterns = [
    ...
    path("health/", include("health.urls")),
    ...
]

```

Finally, we need to update our **k8s/deployment.yaml** to configure Kubernetes to use the new probes:

```

apiVersion: apps/v1
kind: Deployment
...
spec:
  ...
  template:
    ...
    spec:
      containers:
        - name: recommendation-system-app
          ...
          readinessProbe:
            httpGet:
              path: /health/readiness/
              port: 8000
              httpHeaders:
                - name: Host
                  value: "localhost"

```



```
    initialDelaySeconds: 10
    periodSeconds: 5
livenessProbe:
  httpGet:
    path: /health/liveness
    port: 8000
    httpHeaders:
      - name: Host
        value: "localhost"
    initialDelaySeconds: 15
    periodSeconds: 10
```

In this configuration, both the readiness and liveness probes are set up to use the **/health/readiness** and **/health/liveness** endpoint accordingly. The **initialDelaySeconds** and **periodSeconds** are configurable based on how quickly your application starts and how often you want to check its health. We set the Host headers to localhost or any other host in the **ALLOWED\_HOSTS**. Otherwise, Django will reject the request and the probe will fail.

The implementation of these endpoints must be correctly done. If you implement a liveness probe that checks for readiness, it could produce unwanted restarts. The configuration of the probes is also necessary. A high frequency and low timeout could restart the containers prematurely, not allowing the application to recover. You can shoot yourself in the foot with the wrong configuration or probes that don't check what they should do.

## Conclusion

In this chapter, we learned the essential steps and best practices for deploying a Django-based data science API in a scalable, production-ready manner. From configuring Gunicorn as the WSGI server to creating an optimized Docker image for your application, you now understand how to containerize your API and deploy it efficiently.

You have also learned the importance of Kubernetes in orchestrating your deployment, and how to use ConfigMaps, Secrets, StatefulSets, and Probes to ensure high availability, security, and resilience of your application. By following these strategies, you can confidently deploy scalable APIs that can handle production workloads.

As you move forward, consider how these deployment strategies can be applied to other projects, and how Kubernetes can help simplify management at scale. Remember that each deployment environment may require adjustments, and cloud services may offer managed alternatives for some of the tasks described here.

In the next chapter, we will summarize the key lessons from the book and outline the next steps for continued learning and application.

## Questions

1. What is the role of Gunicorn in the deployment process of a Django application?
2. Explain the difference between liveness and readiness probes in Kubernetes. Why are they important?
3. What are the benefits of using multi-stage Docker builds in your deployment process?
4. Describe the difference between ConfigMaps and Secrets in Kubernetes. When should you use each?
5. How does Kubernetes ensure high availability and scalability in your deployment?

[OceanofPDF.com](https://oceanofpdf.com)

## CHAPTER 9

# Final Thoughts and Future Directions

## Introduction

As we reach the end of our journey through building, deploying, and optimizing a Django-based data science API, we will reflect on what we have learned and how these lessons shape our approach to modern web and data science development. We have explored a wide range of topics, from understanding the core of API development to building recommendation systems, implementing caching, optimizing performance, and deploying our API at scale using industry-standard tools like Docker and Kubernetes.

This chapter summarizes the key takeaways from the book, reviews best practices that you can apply to future projects, and looks ahead to emerging trends in web APIs and data science integration. Finally, we will offer some words of encouragement for your continuous learning journey as you continue to refine your skills and adapt to the evolving tech landscape.

## Structure

This chapter covers the following topics:

- Key Takeaways from the Book
- Best Practices in API Development and Data Science
- Future Trends in Web APIs and Data Science
- Final Thoughts and Encouragement for Continuous Learning

## Key Takeaways from the Book

Let us discuss the key takeaways from the book:

- **Building APIs with Django Rest framework**

We began by focusing on the basics of building APIs using the Django Rest Framework (DRF). DRF simplifies the creation of RESTful APIs

by providing a robust set of tools for handling HTTP methods, serialization, authentication, and permissions. It allows developers to create APIs that adhere to REST principles, making it easier to design modular, scalable, and maintainable systems.

You have learned how to define API views, serialize data models, handle authentication methods such as Token and JWT, and secure endpoints with role-based permissions. These fundamentals are key to developing APIs that will serve both front-end applications and external clients.

- **Emphasizing Test-Driven Development (TDD)**

One of the core methodologies we have employed throughout the book is Test-Driven Development (TDD). Writing tests before implementing features ensures that your code is reliable, scalable, and maintainable over time. We used unit tests, functional tests, and integration tests to ensure that each part of our system worked as expected, and we also learned how to create meaningful tests for our recommendation engine.

By following the TDD cycle of red-green-refactor, we ensured that each piece of functionality was correctly implemented and refined without introducing bugs into the system.

- **Data Science Integration and Building a Recommendation System**

In the later chapters, we delved into integrating data science techniques within our API. One of the most practical applications of this integration was the development of a content-based recommendation engine. We explored concepts like cosine similarity, vectorization, and user preference modeling. This blend of API development with data science demonstrated how APIs can serve as the backbone for delivering data-driven insights and personalized user experiences.

Through this process, we also saw the importance of data preparation, using libraries like NLTK to clean and process data for analysis.

- **Optimizing Performance with Caching and Throttling**

API performance is a critical factor for both user satisfaction and scalability. We explored various optimization techniques such as caching, rate limiting, and reducing payload sizes. Caching, in

particular, was implemented using Redis, allowing our API to serve frequently accessed data more quickly while reducing the load on the database.

Additionally, we implemented throttling mechanisms to protect our API from abuse, ensuring that both authenticated and anonymous users are limited in the number of requests they can send within a specified time frame. This helped safeguard our infrastructure and maintain performance under load.

- **Deployment Using Docker and Kubernetes**

In the final technical chapters, we transitioned from development to deployment. We created production-ready Docker images and orchestrated them using Kubernetes. We learned how to configure Gunicorn as a WSGI server to handle multiple requests efficiently and how to manage stateful applications using Kubernetes StatefulSets.

Deploying a Django-based API in a production environment involves ensuring high availability, scalability, and security, and Kubernetes provides the tools to achieve these goals. We also integrated essential features like liveness and readiness probes, ensuring our application could monitor its health and react appropriately to failures or performance bottlenecks.

## **Best Practices in API Development and Data Science**

Let us discuss the best practices in API development and data science:

- **API Design and Documentation**

The foundation of any good API is clear design and thorough documentation. We used the OpenAPI specification with drf-spectacular to generate interactive, auto-updated API documentation. This ensures that API consumers can easily understand how to interact with the endpoints, what parameters are required, and what responses to expect.

In future projects, always prioritize documenting your API as you build it. Documentation not only reduces support inquiries but also serves as the contract between your API and its users.

- **Security First**

Security is paramount in API development. Throughout this book, we emphasized best practices such as using HTTPS, securing sensitive data with Django's built-in authentication and permissions systems, and implementing rate limiting to prevent abuse. Securing API tokens and credentials through environment variables and secrets management in Kubernetes ensures that your API is robust against attacks.

Incorporate security into your development pipeline from the very beginning by implementing authentication, role-based access control, and input validation.

- **Scalability and Performance Optimization**

From caching API responses with Redis to optimizing query performance with pagination and database indexing, we applied techniques that allow APIs to scale. When deploying in production, consider using load balancers and autoscaling mechanisms in Kubernetes to handle fluctuating traffic and ensure high availability.

Scalability is not just about handling more traffic—it is about ensuring that your system remains performant as data size grows and as more users interact with your API.

- **Separation of Concerns and Modularity**

We maintained a separation of concerns by structuring our project with different layers for models, serializers, views, and services. Additionally, using Pydantic models helped decouple our recommendation engine from specific database models, making the system more modular and reusable.

Keep your code modular and organized into well-defined components that can be extended or reused in other projects. This leads to better maintainability, especially in complex systems.

- **Continuous Testing and Monitoring**

Testing should not end when the code is committed. In production environments, it is essential to have monitoring systems in place to continuously check the health and performance of your API. Liveness and readiness probes in Kubernetes helped us monitor the state of the

application while logging and metrics collection provided valuable insights into how the system behaves under load.

## **Future Trends in Web APIs and Data Science**

Some future trends in web APIs and data science are listed as follows:

- **GraphQL and Beyond REST**

While REST APIs are the standard today, alternative paradigms like GraphQL are becoming increasingly popular for their flexibility and efficiency in handling complex queries. GraphQL allows clients to request exactly the data they need, reducing the over-fetching and under-fetching issues often associated with REST APIs.

Looking ahead, consider learning and integrating GraphQL into your toolkit, especially for APIs that serve a wide variety of clients with different data needs.

- **Serverless APIs**

Serverless computing platforms like AWS Lambda and Azure Functions are gaining traction in API development. In a serverless architecture, developers can focus solely on writing the business logic of the API, while the cloud provider automatically handles infrastructure concerns such as scaling, availability, and security.

As serverless architectures evolve, they will become an increasingly appealing option for building scalable, cost-efficient APIs that can handle irregular traffic loads without maintaining traditional servers.

- **Artificial Intelligence Integration**

AI-powered APIs are becoming more prevalent, with APIs providing machine learning models, natural language processing, and predictive analytics as services. For data scientists and engineers, integrating these models into APIs allows for real-time insights, automated decision-making, and personalized user experiences.

The intersection of data science and web APIs will continue to grow, with APIs becoming a crucial delivery mechanism for machine learning models and data-driven insights.

- **Edge Computing and Microservices**

Edge computing, where data is processed closer to where it is generated, is becoming more significant as APIs are deployed on distributed networks. This trend, combined with the rise of microservices, will push APIs to become even more modular and lightweight. APIs may be deployed across multiple nodes, closer to the end-user, reducing latency and improving performance for real-time applications.

## **Final Thoughts and Encouragement for Continuous Learning**

The field of API development and data science is evolving rapidly, and the tools and technologies you have learned throughout this book will continue to grow and change. As you progress in your career or personal projects, continuous learning is essential to stay updated with the latest trends and best practices.

- **Keep Experimenting**

Every API you build and every new challenge you face is an opportunity to learn. Experiment with new frameworks, explore different approaches to data modeling and optimize performance for new types of workloads. Learning is a lifelong journey; the more you practice, the more confident and skilled you will become.

- **Stay Curious**

The world of web development, data science, and APIs is full of exciting innovations. Stay curious about emerging technologies like GraphQL, serverless computing, and AI-powered APIs. Follow thought leaders, join communities, and read industry publications to stay informed about the latest developments.

- **Contribute to the Community**

Contributing to open-source projects or sharing your knowledge through blog posts, tutorials, or talks can accelerate your learning. It also helps the broader development community by advancing the collective understanding of best practices and new technologies.



## Conclusion

We have covered a lot of ground in this book, from building APIs with Django and Django Rest Framework to integrating data science techniques and deploying APIs in production environments. You have learned the importance of security, scalability, and performance, and how to use modern tools, such as Docker, Kubernetes, and Redis to manage complex systems.

As you continue on your path, remember that the best developers are always learning, adapting, and improving their craft. The skills you have gained through this book will serve as a foundation, but there is always more to explore. Whether you are building APIs, integrating data science models, or managing deployments at scale, your next project will benefit from the knowledge and best practices you have acquired.

Stay curious, keep learning, and continue building amazing APIs!

[OceanofPDF.com](https://oceanofPDF.com)

# Index

## Symbols

`__call__` [117](#)

`@permission_classes` [128](#)

## A

AAA, pattern

Act [24](#)

Arrange [24](#)

Assert [24](#)

API, architecture [60](#), [61](#)

API, authentications [62-67](#)

APIClient [64](#), [65](#)

API, database [80](#)

API development/Data Science, best practices [201](#), [202](#)

API Development/Data Science, thoughts [203](#), [204](#)

API, endpoints [61](#)

API-First Design [34](#)

API-First Design, process [35](#)

API, process [80-86](#)

API Views, testing [38-42](#)

Asynchronous Data Process, testing [97](#)

## B

Background Task, processing [100](#)

Background Task Processing [90](#)

Background Task Processing, components

API Server [90](#)

Task Queue [90](#)

Worker [90](#)

Book, key takeaways [200](#), [201](#)

## C

Caching [163](#), [164](#)

Celery [91](#)

Celery, features

Broker, supporting [91](#)

Distributed Task, processing [91](#)

Result, backend [91](#)

Task, scheduling [91](#)

Task, workflows [91](#)

- Celery Results, backend [96](#)
- Celery Tasks, utilizing [93](#), [94](#)
- Celery With Django, configuring [95](#), [96](#)
- clean\_text [139](#)
- Cloud File Storage, configuring [100-105](#)
- combine\_attributes [148](#)
- COPY Command [176](#)
- Cosine Similarity, role [135](#), [136](#)
- CountVectorizer [147](#)
- CRUD API [36](#)
- CRUD API, attributes
  - Genres [36](#)
  - ID [36](#)
  - Title [36](#)
- CRUD Operations, developing [47-50](#)
- CSV Project, optimizing [76-79](#)
- Curl (client URL) [19](#)
- Custom Permissions, breakdown
  - %(app\_label)s [127](#)
  - %(model\_name)s [127](#)
  - view\_ [127](#)
- Custom Permissions, optimizing [125-129](#)

## D

- Data APIs [129](#)
- Data APIs, sections
  - API Keys, managing [132](#)
  - Authentication Methods, securing [130](#)
  - CORS, handling [130](#)
  - data limit, exposing [131](#)
  - dependencies, updating [132](#)
  - Enforce HTTPS [129](#)
  - input data, validating [130](#)
  - Monitor API, log [131](#)
  - penetration, testing [132](#)
  - rate limit, implementing [130](#)
- Data Structures, categories
  - Dictionary [5](#)
  - Lists [5](#)
  - Set [5](#)
  - Tuple [4](#)
- Designing Data Models [55](#)
- Designing Data Models, columns
  - country [56](#)
  - extra\_data [56](#)
  - release\_year [56](#)
- Designing Data Models, extending [56-59](#)
- detect\_q\_strings [139](#)
- divide\_function [9](#)

- Django [21](#)
- Django Authentication System [113](#), [114](#)
- Django Groups/Permissions [119](#)
- Django, installing [22](#)
- Django, key points
  - Request, handling [21](#)
  - Request, processing [21](#)
  - Response, generating [21](#)
  - Response, sending [22](#)
- Django Models, building [25](#), [26](#)
- Django Permissions, classes
  - AllowAny [119](#)
  - IsAdminUser [119](#)
  - IsAuthenticated [119](#)
  - IsAuthenticatedOrReadOnly [119](#)
  - IsOwner [119](#)
- Django REST Framework (DRF) [43-46](#)
- Docker [92](#)
- Docker Compose [92](#)
- Docker Compose, optimizing [92](#)
- Dockerfile [172](#)
- Dockerfile, dependencies
  - manage.py [177](#)
  - pyproject.toml [177](#)
  - README.md [177](#)
  - requirements.lock [177](#)
- Dockerfile, instructions
  - CMD [173](#)
  - COPY [173](#)
  - ENV [173](#)
  - EXPOSE [173](#)
  - FROM [173](#)
  - RUN [173](#)
  - WORKDIR [173](#)
- Dockerfile, stages [174](#), [175](#)
- Dockerfile, variables [175](#), [176](#)
- DRF Authentication, methods
  - Basic Authentication [120-122](#)
  - JWT (JSON Web Token) [123-125](#)
  - Token Authentication [121-123](#)
- DRF, classes
  - CreateAPIView [51](#)
  - DestroyAPIView [51](#)
  - ListAPIView [51](#)
  - RetrieveAPIView [51](#)
  - UpdateAPIView [51](#)
- DRF, key features
  - Authentication, permissions [23](#)
  - Browsable API [23](#)
  - Serialization [23](#)

- Testing [23](#)
- Throttling [23](#)
- DRF Serializers, types
  - HyperlinkModelSerializer [43](#)
  - ModelSerializer [43](#)
  - Serializer [43](#)
- DRF, tests
  - End-to-End Tests [24](#)
  - Integration Tests [24](#)
  - Performance Tests [24](#)
  - Unit Tests [24](#)

## F

- fibonacci [10](#)
- F-Strings [4](#)

## G

- GeneralFileUploadSerializer [82](#)
- Git [30](#), [31](#)
- Git, methodologies
  - Collaborative [32](#)
  - Content-Based [32](#)
  - Context-Aware [32](#)
  - Hybrid Systems [32](#)
  - Knowledge-Based [32](#)
- Gunicorn [169](#)
- Gunicorn, configuring [169-172](#)
- Gunicorn, functions
  - ALLOWED\_HOSTS [170](#)
  - CORS\_ALLOWED\_ORIGINS [170](#)
  - DEBUG [170](#)

## H

- HTTP, classes
  - 1xx [16](#)
  - 2xx [16](#)
  - 3xx [16](#)
  - 4xx [16](#)
  - 5xx [16](#)
- HTTP, headers
  - Entity [18](#)
  - General [18](#)
  - Request [18](#)
  - Response [18](#)
- HTTP Headers, lists [18](#)
- HTTP, methods [15](#)
- HTTP Protocol [114](#)

- HTTP Protocol, method
  - Basic Authentication [114](#)
  - JWT [114](#)
  - OAuth [114](#)
  - Token Authentication [114](#)
- HTTP, status codes [17](#)
- Hypertext Transfer Protocol (HTTP) [15](#)

## I

- Image Registry [179](#)
- IP-Based Throttling [165](#)

## J

- JavaScript Object Notation (JSON) [11](#)
- JWT (JSON Web Token) [123-125](#)

## K

- K8s, applications
  - Cluster [180](#)
  - ConfigMap [181](#)
  - Deployment [181](#)
  - Ingress [181](#)
  - Node [180](#)
  - Pods [180](#)
  - ReplicaSets [181](#)
  - Scheduler [180](#)
  - Secrets [181](#)
  - Services [181](#)
  - StatefulSets [182](#)
- KeyError [8](#)
- kubectrl [184-186](#)
- Kubernetes Cluster, configuring [183-186](#)
- Kubernetes (K8s) [179](#)

## L

- Large Data Sets [106](#)
- Large Data Sets, components
  - Chunk Worker [106](#)
  - File Splitter Worker [106](#)
- Large Data Sets, handling [107-111](#)
- Large Data Sets, patterns
  - Canvas [107](#)
  - Chains [107](#)
  - Chords [107](#)
  - Groups [107](#)
- Liveness Probes [195](#)

Local Environment, preparing [136](#), [137](#)

## M

Middleware [114-116](#)

Middleware, architecture [116](#), [117](#)

Middleware, setting up [118](#)

MinIO [103](#)

Model-View-Template (MVT) [60](#)

Multi-Stage [174](#)

MVT Service, layers [60](#)

## N

Natural Language Toolkit (NLTK) [138](#)

NLTK, installing [138-140](#)

## O

OpenAPI [21](#), [155](#)

OpenAPI, elements

Methods [156](#)

Parameters [156](#)

Path [156](#)

Responses [156](#)

Security [156](#)

OpenAPI Endpoint, documenting [159-162](#)

OpenAPI, integrating

Documentation, generating [156](#)

Documentation, updating [156](#)

Inquiries, reducing [156](#)

interactive, documentation [156](#)

OpenAPI, setting up [157](#)

## P

pathlib [11](#)

Payload Size, reducing [164](#), [165](#)

Performance Optimization Techniques [162](#)

PostgreSQL [186-188](#)

PostgreSQL, concepts

PersistentVolume [187](#)

PersistentVolumeClaim [187](#)

PostgreSQL Databases, command

build-essential [176](#)

curl [176](#)

libpq-dev [176](#)

PreferencesSerializer [68](#)

pytest [27](#)

Pytest [27](#), [28](#)

- Python [1-3](#)
- Python, characteristics
  - Class/Objects [7](#)
  - Data Structures [4](#)
  - Decorators [10](#), [11](#)
  - Duck, typing [9](#)
  - Error, handling [8](#), [9](#)
  - Functions [6](#), [7](#)
  - Generators [9](#), [10](#)
  - Linting [14](#)
  - Modules [11](#), [12](#)
  - Polymorphism [9](#)
  - Type Hints [14](#)
  - Variables, optimizing [3](#)
- Python Environment, setting up [13](#)
- Python Web Server Gateway Interface (WSGI) [168](#), [169](#)

## R

- RDF (Resource Description Framework) [56](#)
- Readiness Probes [195-197](#)
- Recommendation Engine [134](#), [135](#)
- Recommendation Engine, algorithms [135](#)
- Recommendation Engine, approach
  - Collaborative, filtering [135](#)
  - Content-Based, filtering [135](#)
  - Hybrid Methods [135](#)
- Recommendation System [141-143](#)
- Recommendation System, implementing [145-148](#)
- Recommendation System/Unit Tests, validating [143](#), [144](#)
- Redis [163](#)
- Refactor API [98](#)
- Representational State Transfer (REST) [19](#)
- REST, principles
  - Cacheability [20](#)
  - Client-Server [20](#)
  - Code on Demand [20](#)
  - Layered, system [20](#)
  - Stateless [20](#)
  - Uniform, interface [20](#)

## S

- Secrets Operations [189-193](#)
- SecurityMiddleware [115](#)
- Services, types
  - ClusterIP [181](#)
  - LoadBalancer [181](#)
  - NodePort [181](#)
- SPARQL [73](#), [74](#)



SPARQL, endpoints [75](#)  
SPARQL, preventing [75](#)

## T

TDD, cycle  
    GREEN [25](#)  
    RED [25](#)  
    REFACTOR [25](#)  
TDD With Code, refactoring [50-52](#)  
Test Driven Development (TDD) [25](#)  
Token-Based Authentication [36-38](#)  
TypeError [3](#)

## U

UserMoviePreferences [58](#)  
user\_preferences [71](#)  
UserPreferencesView [72](#)  
user\_watch\_history [71](#)

## V

Version Control System (VCS) [30](#)

## W

Web APIs/Data Science, best practices [202](#), [203](#)  
Wikidata [56](#)  
Wikidata, properties  
    wd:Q11424 [76](#)  
    wdt:P31 [76](#)  
    wdt:P57 [76](#)  
    wdt:P136 [76](#)  
    wdt:P495 [76](#)  
    wdt:P577 [76](#)

[OceanofPDF.com](http://OceanofPDF.com)