# Django REST Framework Basics


**SoftUni Team**

**Technical Trainers**

Software University

SoftUni

Software University

https://softuni.org

# Have a Question?

Software University

sli.do

#python-web

# Table of Contents




# RESTful APIs

## Introduction, Key Principles

# What is API?

- An **API** stands for **Application Programming Interface**
- **Serves** as an **intermediary**
- **Defines** how **different software components** or **systems** can **interact** with each other
- **Outlines** the **methods** and **protocols** through which one **system** can **communicate** with **another**

# What is REST?

- **REST** stands for **Representational State Transfer**
- It is an **architectural style** for **designing** networked **applications**
- **RESTful APIs** are built based on the **principles** of **REST**, which **emphasizes**:
  - The **stateless client-server relationship** and **uniform interface**

# Introduction to RESTful APIs

- **RESTful APIs** have become a **cornerstone** of **modern web** development
  - Enabling **seamless communication** between **different** software **systems** over the internet
- Understanding the **concept** and **principles** of **RESTful APIs** is **essential** for any developer entering the world of **web development**

# Key Principles of RESTful APIs

- **Client-Server Architecture**
  - RESTful APIs follow a **client-server architecture**
  - The **client sends requests** to the **server**, and the **server processes** those **requests** and **sends back responses**
- **Statelessness**
  - RESTful APIs are **stateless**
  - Each **request** from a **client** to the **server** must **contain all** the **necessary information** to **understand** and **process** the **request**
  - The server does **not store** any **client state between requests**

# Key Principles of RESTful APIs

- **Uniform Interface**
  - RESTful APIs have a **uniform interface**
  - They **follow** a **set** of **standard rules** for **communication** between the **client** and the **server**
  - This includes using **standard HTTP methods** (**GET**, **POST**, **PUT**, **DELETE**) to **perform actions** on **resources** and using **standard HTTP status codes** to indicate the **outcome** of a **request**

# Key Principles of RESTful APIs

- **Resource-Based**
  - In RESTful APIs, **everything** is **treated** as a **resource**, which can be **accessed** and **manipulated** using **standard HTTP methods**
  - Resources are **identified** by **unique URIs** (**Uniform Resource Identifiers**), and clients **interact** with these resources **through** the **API**
- **Representation**
  - **Resources** in RESTful APIs are **represented** in different **formats**
    - Such as **JSON** (**JavaScript Object Notation**) or **XML** (**eXtensible Markup Language**)

# Benefits of RESTful APIs

- **Scalability**
  - RESTful APIs are **highly scalable**
- **Simplicity**
  - Thanks to the **uniform interface** and **stateless nature**
- **Flexibility**
  - Support of **wide range** of **clients** like web **browsers**, **mobile** devices, and **IoT** devices
- **Interoperability**
  - Can be used **across different platforms** and **technologies**

Django
REST

# Django REST Framework

Introduction

# What is Django REST Framework?

- **Django REST Framework** (**DRF**) is an **extensive** and **adaptable** **toolkit** designed for **constructing** **Web APIs** using Django
- **Key Features** of **Django REST Framework**
  - **Powerful** Toolkit
  - **Flexible** Design
  - **Standardized** Requests and Responses
  - **Serialization**
  - **Extension** Mechanisms

# Benefits of Django REST Framework

- **Accelerated Development**
  - **Streamlines** API **development** by providing a **comprehensive set** of **pre-built components**
- **Scalability**
  - The **modular architecture facilitates** the development of **scalable APIs**
- **Community Support**
  - **Benefits** from a vibrant and **engaged community** of developers

# Reasons to Use DRF

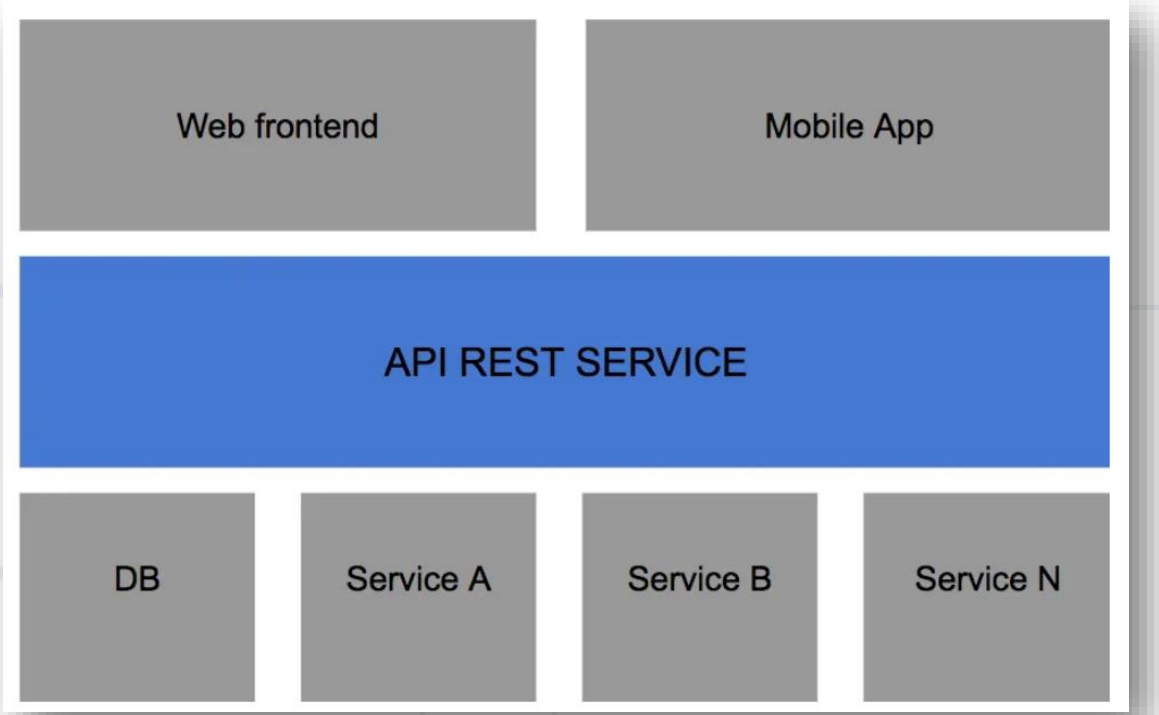
- **Web Browsable API**
  - Offers a **user-friendly**, browsable API that **enhances usability** and **simplifies interaction** with APIs
- **Authentication Policies**
  - Provides **support** for **authentication**, including built-in packages for **OAuth1a** and **OAuth2** **authentication protocols**

# Reasons to Use DRF

- **Serialization Flexibility**
  - Offers **powerful serialization capabilities** that **support both** Object-Relational Mapping (**ORM**) and **non-ORM** data **sources**
- **Trusted by Industry Leaders**
  - **Widely used** and **trusted** by **numerous** companies and **organizations**, including **Mozilla**, **Red Hat**, **Heroku**, and **Eventbrite**

# Django REST and RESTful APIs

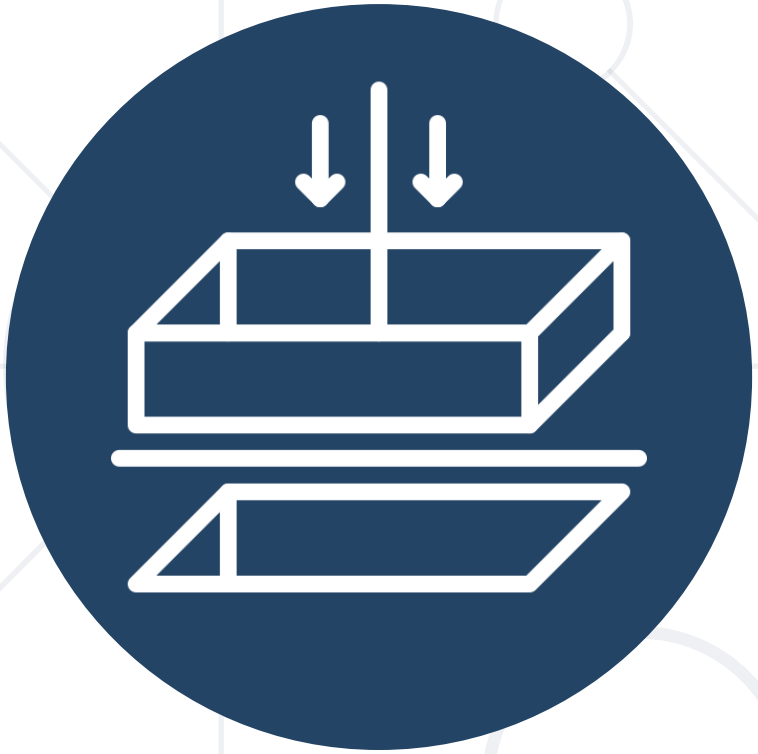
- **DRF simplifies** the creation of **RESTful APIs**
  - Serving as a means to **facilitate** the **exchange** of **data** between a **user interface** and a **database**



# RESTful API Structure

- **Endpoints** (**URLs**) **define** the **structure** of the **API**
  - **Users access** these **endpoints** using **HTTP methods** (GET, POST, PUT, and DELETE)

| Endpoint | GET | POST | PUT | DELETE |
|---|---|---|---|---|
| /books/ | Show all books | Add a new book | Update all books | Delete all books |
| /books/<id>/ | Show <id> | N/A | Update <id> | Delete <id> |



# Requirements and Installation

# Requirements

- To use the Django REST Framework, you need
  - **Python** (version 3.6 or higher)
  - **Django** (version 3.0 or higher)
- It is **recommended** to use the **officially supported** and **latest versions** of **Python** and **Django** for **compatibility** and to **access** the **latest features** and **improvements**
- There are **optional dependencies** that you may choose to install based on your specific requirements
  - ***Markdown***, ***Pygments***, ***django-filter***, ***django-guardian***

# Installation and Setup

- To **install** Django REST, you can use the **pip** command

```
pip install djangorestframework
```

- You need to **add** it to the **INSTALLED_APPS** setting
- **Create** an **app** and **add** it also to the **INSTALLED_APPS** setting
- **Create** and **Include** the app's **urls** to the project's **urlpatterns**

```
INSTALLED_APPS = [
    ...
    'rest_framework',

    'books_api',
]
```

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

# Creating a Simple RESTful API

## Books API – Live Demo (Lab)

# Creating the Book Model

- After installing the Django REST Framework and setting it up, we will **create** our **Book** model and **migrate** it to the **database**

```
from django.db import models

# Create your models in the models.py file

class Book(models.Model):
    title = models.CharField(max_length=20)
    pages = models.IntegerField(default=0)
    description = models.TextField(max_length=100, default="")
    author = models.CharField(max_length=20)
```

# DRF Serializers

- **Serializers** in DRF **facilitate** the **conversion** of **complex data structures**, such as Django **model instances**, into **native Python datatypes**
- These **data types** can then be **effortlessly rendered** into various **formats**, including **JSON**, **XML**, and more
  - They **simplify data transmission** and **consumption** across different platforms
- **Serializers** offer **deserialization capabilities**, enabling **parsed data** to be seamlessly **converted back** into **complex types**
  - Ensures **compatibility** and **consistency** in **data** handling

# Create a Serializer for the Book Model

- Create a **corresponding serializer** for the Book model to **convert book instances into JSON format** and **vice versa**

```
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = '__all__'
```

# DRF APIView

- **APIView** in DRF is a **class-based view** that allows developers to **define custom API endpoints** with **fine-grained control** over their behavior
- It serves as the **foundation** for building **RESTful APIs**, offering **flexibility** and **extensibility** to handle various **HTTP requests** such as **GET**, **POST**, **PUT**, and **DELETE**
  - Providing own **methods** like `get()`, `post()`, `put()`, and `delete()` to **handle** those **HTTP requests** and perform corresponding **actions** on **data**

# DRF Request and Response objects

- The **Request object** in DRF **encapsulates** **incoming** HTTP **requests**, providing **access** to **request data**
  - Such as **headers**, **parameters**, and **payloads**
- It offers convenient **methods** and **attributes** for **extracting** and **processing** **data** sent by clients to the server
  - These **attributes** allow developers to **access** **request body data**, **query parameters**, and **authenticated user** information

# DRF Request and Response objects

- The **Response object** facilitates the **generation** of **HTTP responses**, allowing developers to construct and **return** back to the client:
  - **Structured data** as **JSON** or **XML**
  - **Appropriate status codes**, **headers**, and **content types**
- This abstraction **simplifies** the **handling** of **request and response cycles**
  - Promoting **efficient communication** between **clients** and **servers** in **DRF-powered APIs**

# Create the ListBooksView

- The **ListBooksView** will handle **GET** and **POST** requests on **`localhost:8000/api/books/`**

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Book
from .serializers import BookSerializer

class ListBooksView(APIView):
    def get(self, req):
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response({"books": serializer.data})
```

# Create the URL

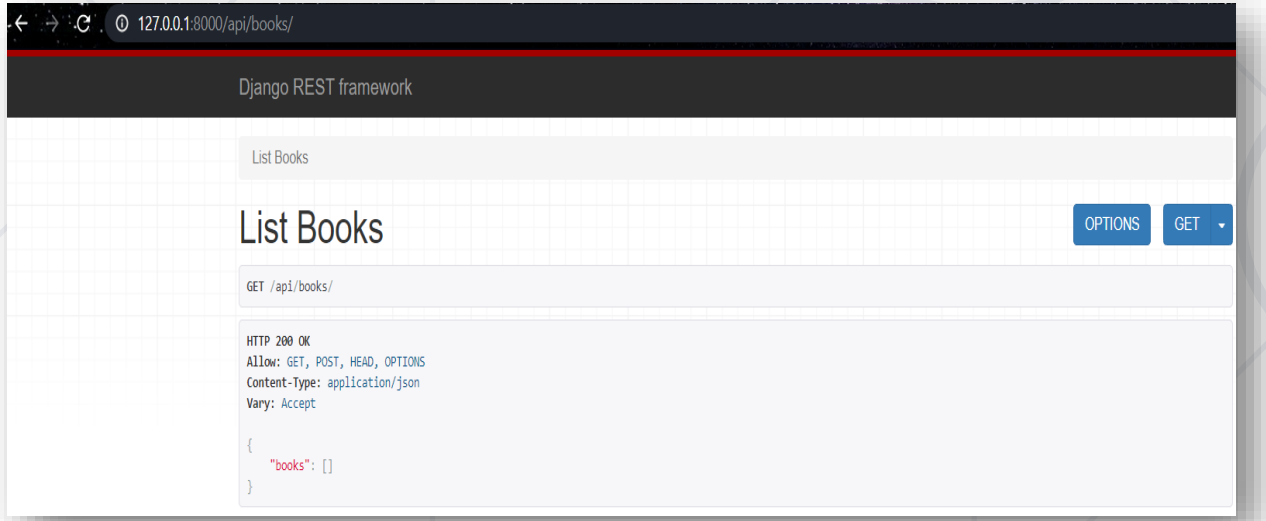
- Now, you need to add the **URL pattern**

```
from django.urls import path
from books.views import ListBooksView

urlpatterns = [
    path('books/', ListBooksView.as_view(), name="books-all"),
]
```

# Testing the API

- To run the API, we use the **same command** as for regular Django projects

```
python manage.py runserver
```



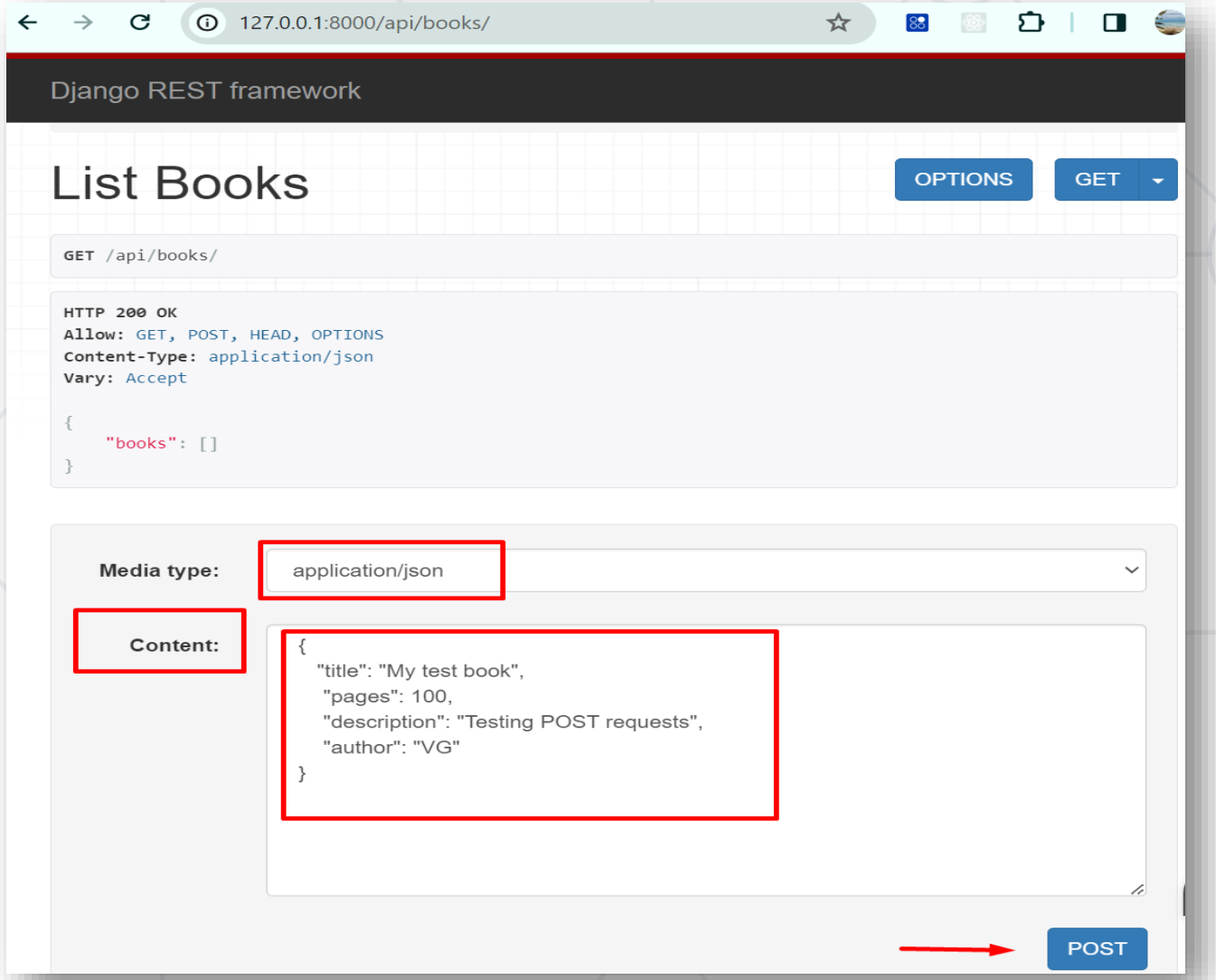
# Implementing the POST Request

```
class ListBooksView(APIView):
    ...
    def post(self, req):
        serializer = BookSerializer(data=req.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

Returns a status code 201 upon success

Returns a status code 400 upon errors

# POST Requests




Data shall be passed as a JSON object

# POST Requests

List Books

## List Books

OPTIONS GET

```
POST /api/books/
```

```
HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 3,
    "title": "My test book",
    "pages": 100,
    "description": "Testing POST requests",
    "author": "VG"
}
```

The successful POST request returns the record with an auto-generated id in the response

## List Books

OPTIONS GET

```
POST /api/books/
```

```
HTTP 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "title": [
        "Ensure this field has no more than 20 characters."
    ],
    "author": [
        "This field is required."
    ]
}
```

The unsuccessful POST request returns the errors and a status code of 400 in the response

# Create a DetailBookView

- The view will handle **GET**, **PUT**, and **DELETE** methods

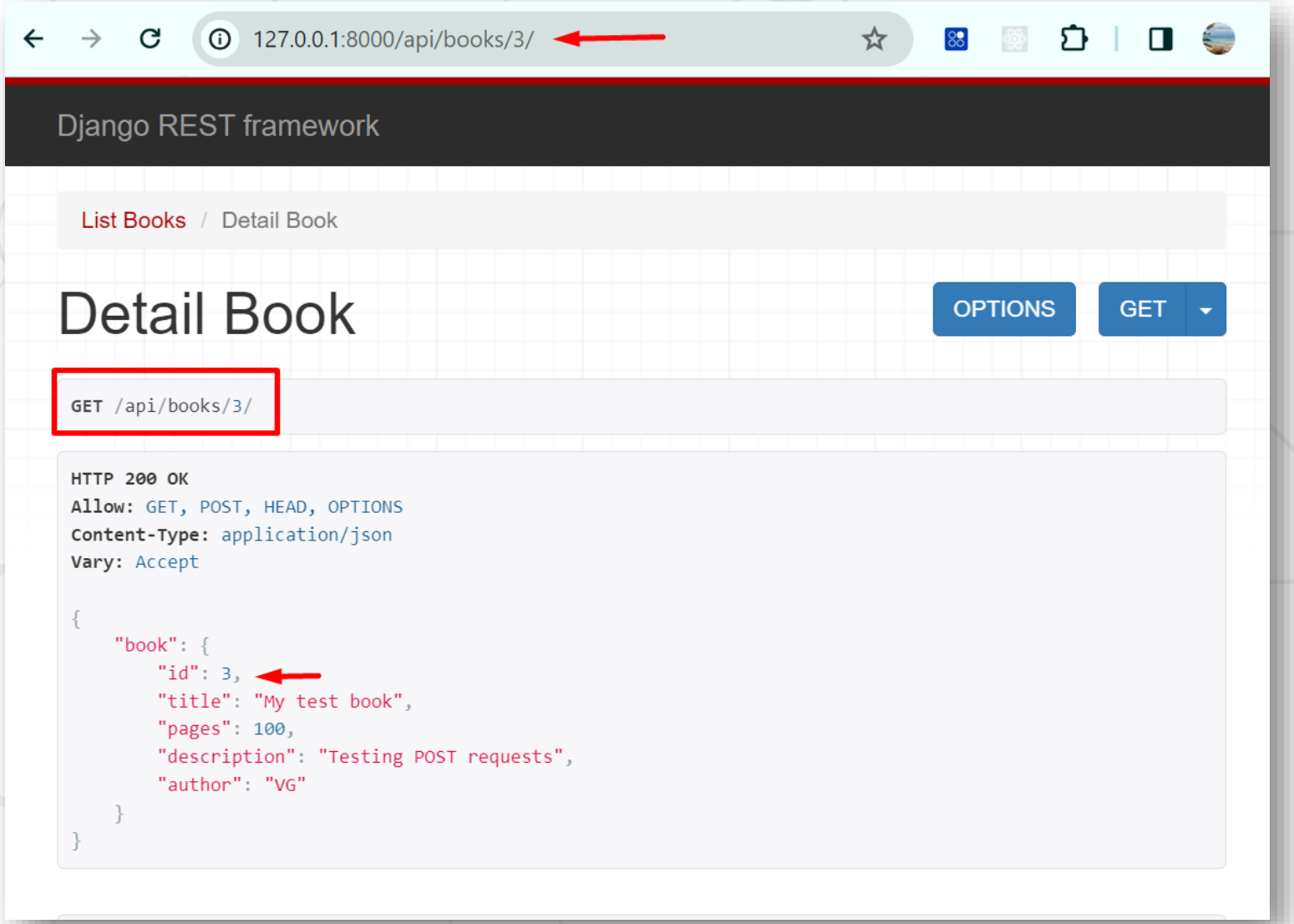
```
class DetailBookView(APIView):
    def get(self, req, id):
        book = Book.objects.get(pk=id)
        serializer = BookSerializer(book)
        return Response({"book": serializer.data})

    def put(self, req, id):
        book = Book.objects.get(pk=id)
        serializer = BookSerializer(book, data=req.data)
        if serializer.is_valid():
            serializer.save()
        # TODO: Return a Response
    # TODO: Implement the DELETE
```

# Create the URL

- On `localhost:8000/api/books/{id}` you will be able to **GET**, **Update**, and **Delete** a specific book

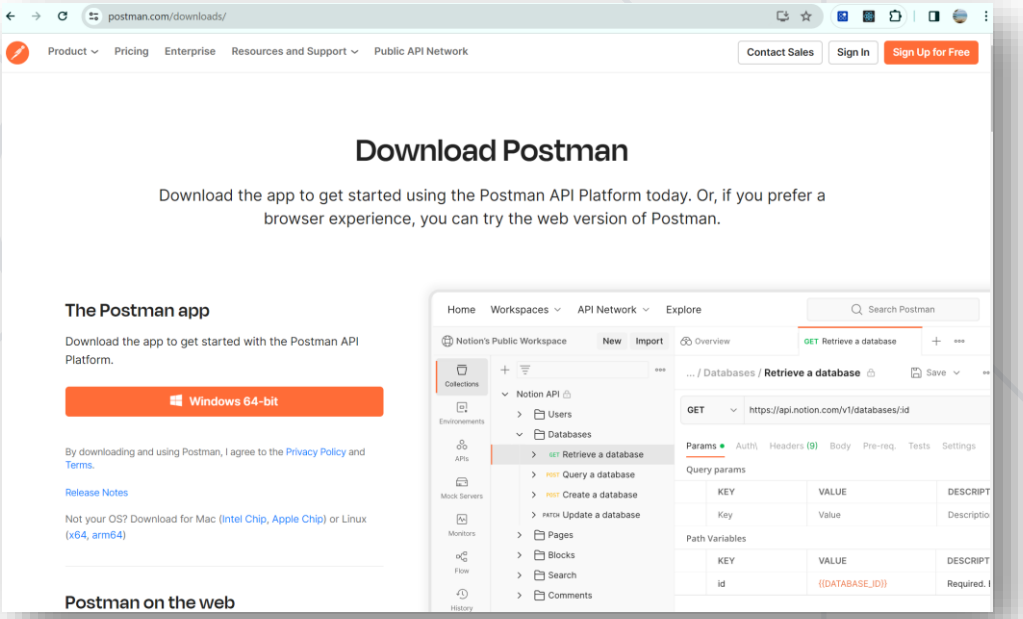
```
from django.urls import path
from books.views import ListBooksView, DetailBookView

urlpatterns = [
    path('books/', ListBooksView.as_view(), name="books-all"),
    path('books/<int:id>/', DetailBookView.as_view(), name="book-details")
]
```

# Check the URL

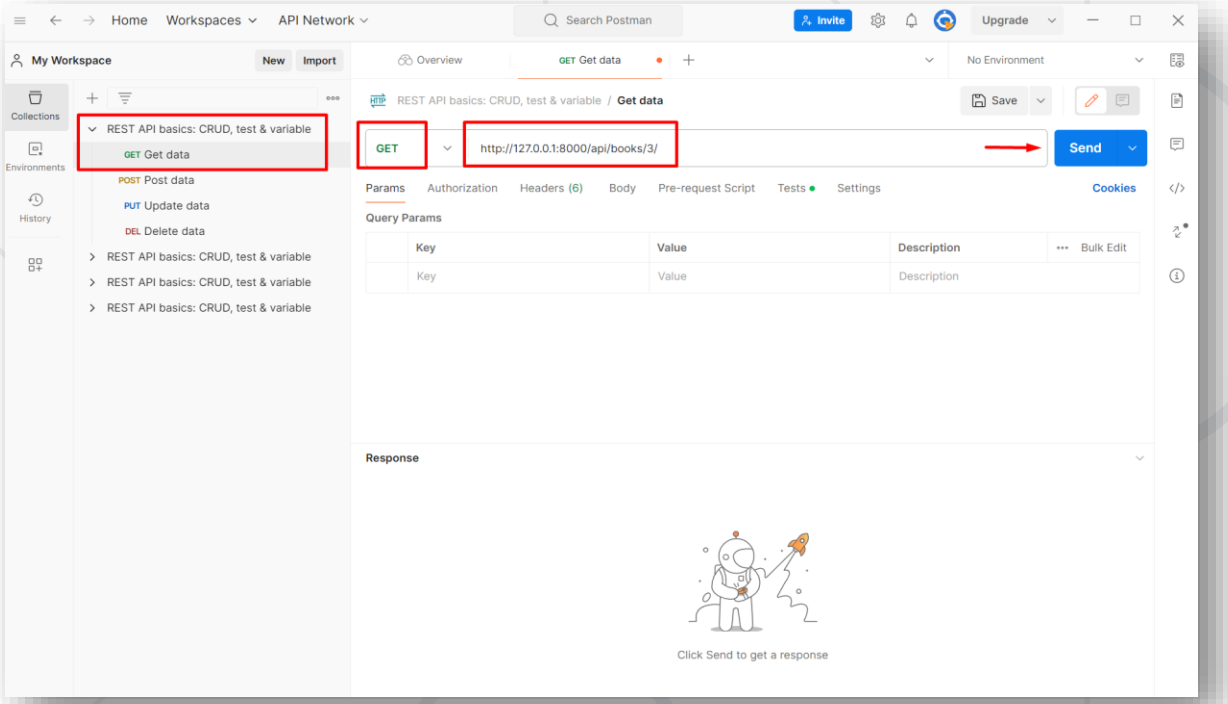


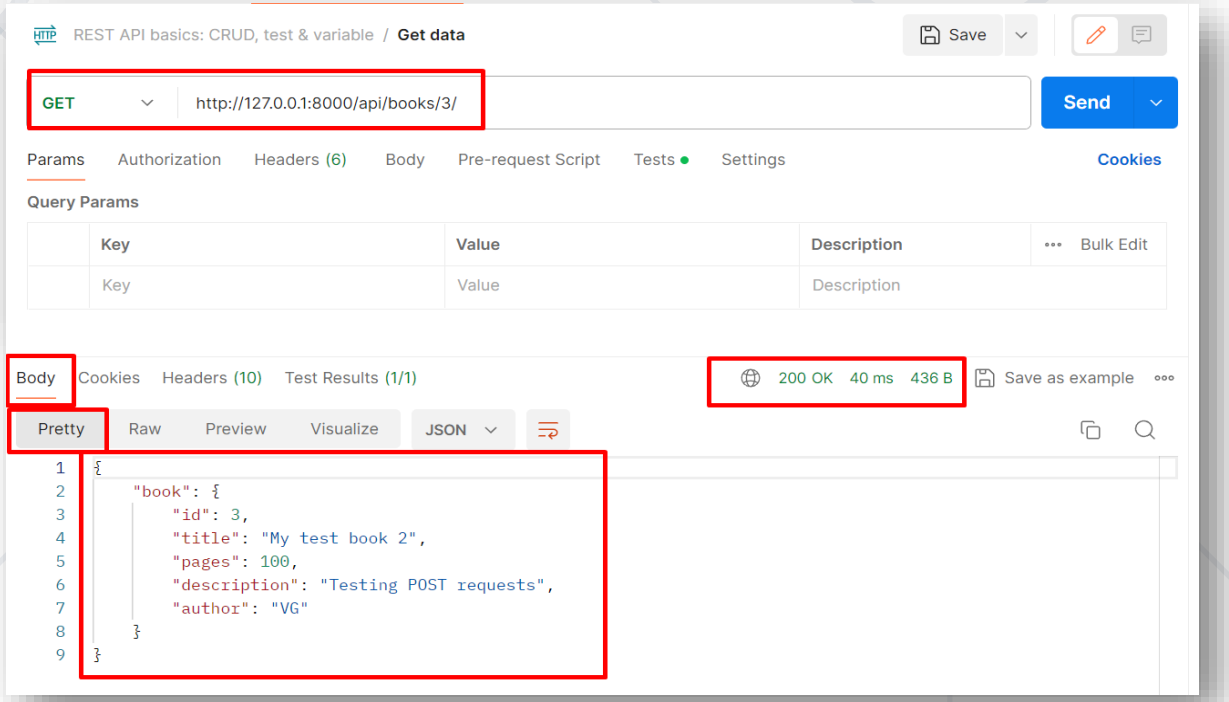
# HTTP Requests via Postman

- Optionally, you can use the **Postman API** to send **HTTP Requests** and get **HTTP Responses**
  - You need to download the **Postman Desktop Agent** from ***https://www.postman.com/downloads/***
  - **Install** and **run** the application
  - Create an **account** or **sign in**

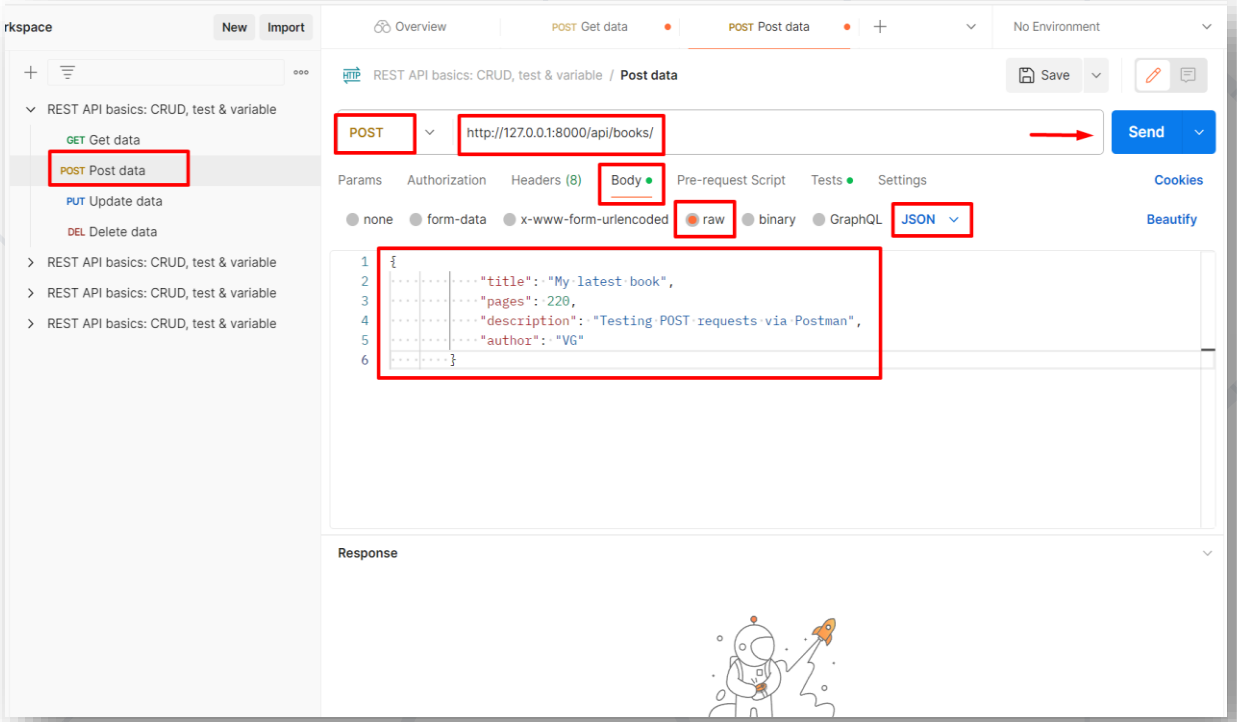


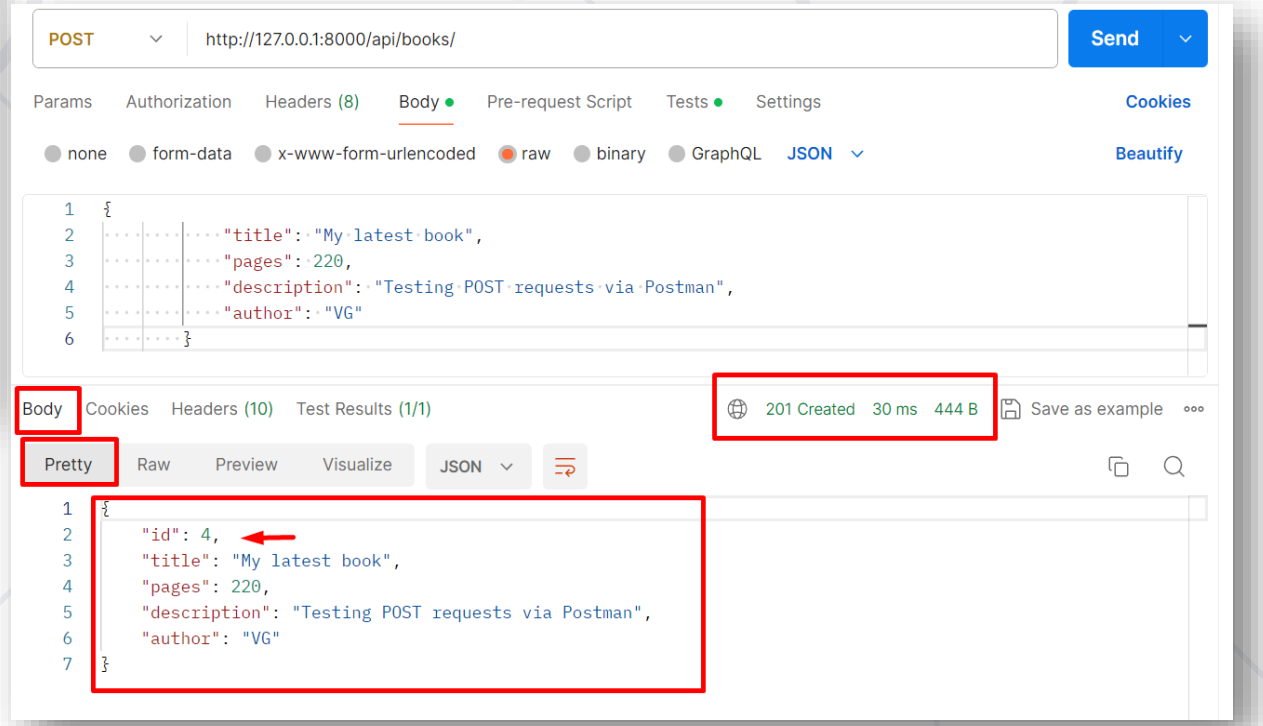
# HTTP Requests via Postman - GET Request



# HTTP Requests via Postman - POST Request

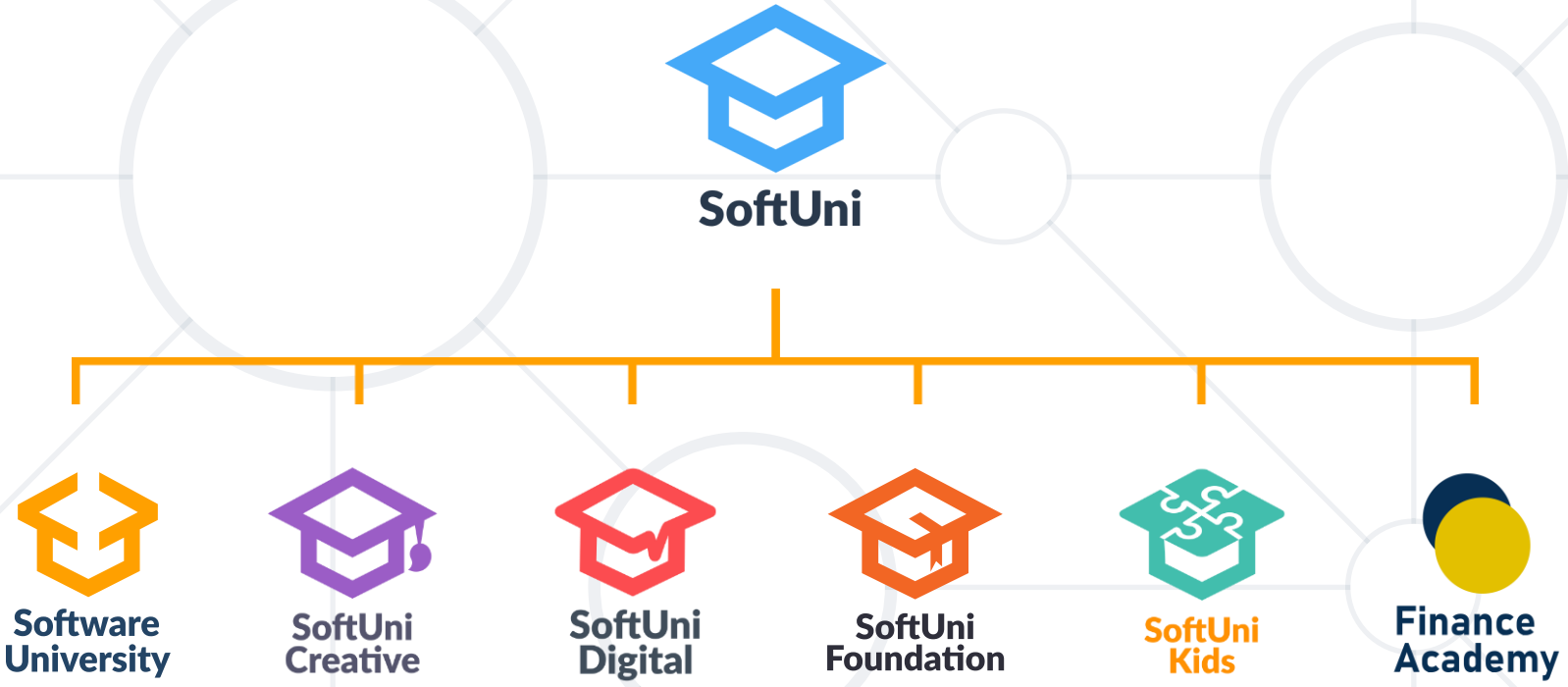


# Summary

- **RESTful APIs**
- Django **REST** Framework
  - **Serializers**
  - **APIView**
  - Request and Response Objects
- **Postman** API

# Questions?



# SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank
Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR
Serving the high achievers

AMBITIONED

PHAR VISION

# Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

# License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg