Django REST Framework Advanced

SoftUni Team

Technical Trainers

Software University

https://softuni.org

Have a Question?

Software University

sli.do

#python-web

Table of Contents

Advanced Serialization

Nested Serializers

What are Nested Serializers?

- **Nested serializers** in Django Rest Framework (DRF) allow you to **serialize** and **deserialize** **complex nested data structures**
- **Useful** when:
  - There are **relationships** between **models**
  - You need to include **related data** in **API responses** or handle **nested data** in **API requests**

Nested Model Serializers

- **Nested model serializers** are typically used in scenarios where you have **models** with **relationships** between them
- If you have a **Parent** model and a **Child** model where each **parent** can have **multiple children**, you might want to **include information** about the **children** when **serializing** a **parent instance**

Create the Related Models

```
# models.py

from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, on_delete=models.CASCADE,
related_name="books")
```

Create Model Serializers

```python
# serializers.py
from rest_framework import serializers
from .models import Author, Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ['title']

class AuthorSerializer(serializers.ModelSerializer):
    books = BookSerializer(many=True, read_only=True)

    class Meta:
        model = Author
        fields = ['name', 'books']
```

Nest the BookSerializer within the AuthorSerializer

Nested Model Serializers

- In the example, the `AuthorSerializer` includes a **nested representation** of the **books** **associated** with each **author**
- When you **serialize** an **author instance**, it will **include** the **titles** of **all** the **books** written by that author

Generic API Views

Generic API Views

- **Generic API views** in DRF provide a set of **pre-built views** that help you **quickly create APIs** for
  - **common CRUD** operations
  - **without** having to **write** a lot of **boilerplate code**
- These **views** are **designed** to work with Django **models** and provide a **standardized** way to **interact** with your **data** through **HTTP methods** like
  - **GET**, **POST**, **PUT**, **PATCH**, and **DELETE**

Generic API Views

- **ListAPIView**:
  - **Retrieves** a **list** of **objects** from the database
- **RetrieveAPIView**:
  - **Retrieves** a **single object** by its **primary key**
- **CreateAPIView**:
  - **Creates** a **new object**
- **UpdateAPIView**:
  - **Updates** an **existing object** by its **primary key**

Generic API Views

- DestroyAPIView:
  - **Deletes** an **existing object** by its **primary key**
- ListCreateAPIView:
  - **Combines list** and **create functionalities** into a **single view**
- RetrieveUpdateAPIView:
  - **Combines retrieve** and **update functionalities** into a **single view**

Generic API Views

- RetrieveDestroyAPIView:
  - **Combines retrieve** and **destroy functionalities** into a **single view**
- RetrieveUpdateDestroyAPIView:
  - **Combines retrieve**, **update**, and **destroy functionalities** into a **single view**

More about Generic API Views: *django-rest-framework.org/api-guide/generic-views/#concrete-view-classes*

ListCreateAPIView - Example

- Define a view that **handles** HTTP **GET** (**list**) and **POST** (**create**) **requests** related to the **Author model**

```python
# views.py
from rest_framework import generics
from .models import Author
from .serializers import AuthorSerializer

class AuthorList(generics.ListCreateAPIView):

    queryset = Author.objects.all()
    serializer_class = AuthorSerializer
```

Specify the **queryset** of Author objects to be used for listing

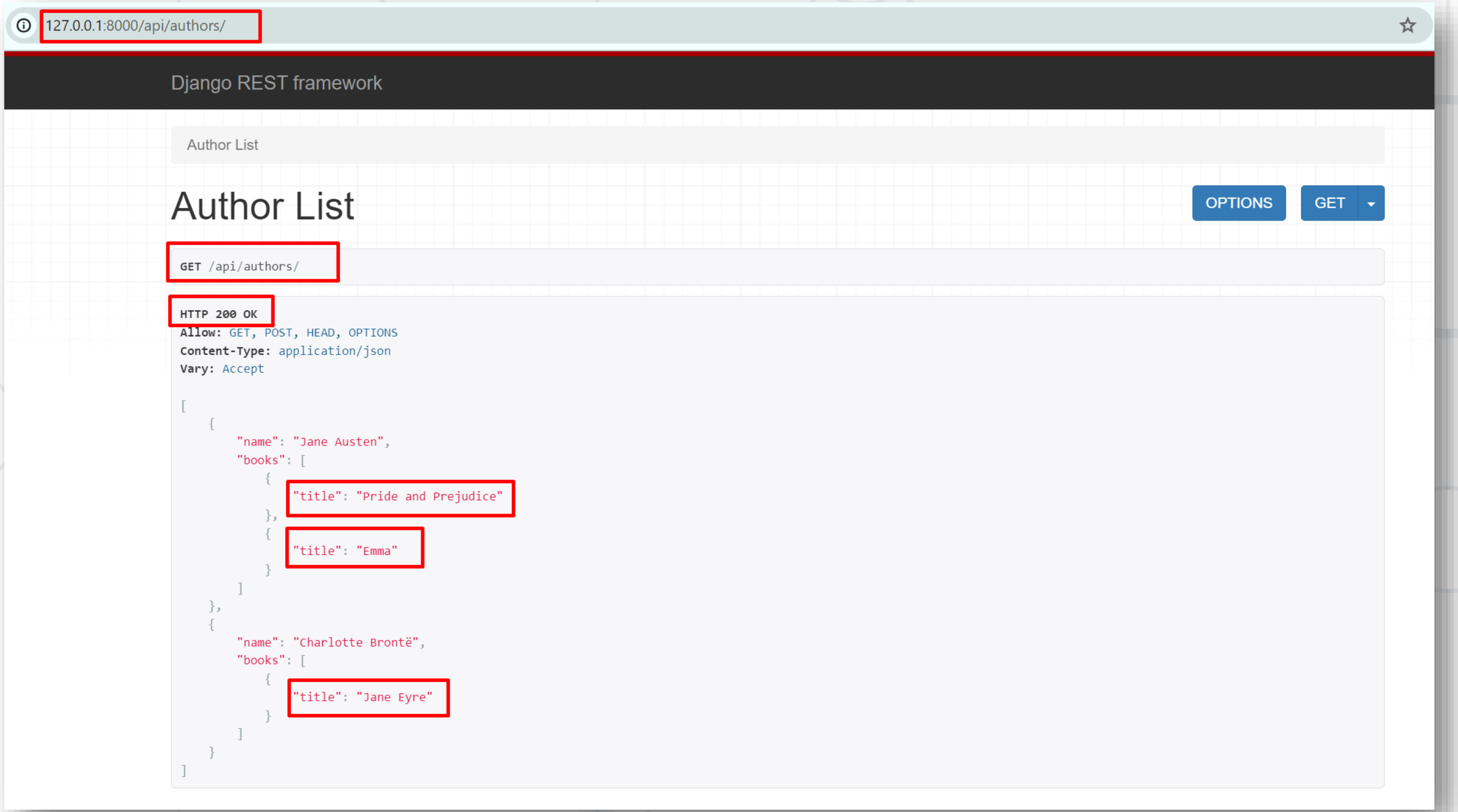
Specify the **serializer class** to be used for serializing/deserializing Author objects

AuthorList GET Request - Example

- **Access** the **endpoint** served by `AuthorList`
  - The **JSON response includes** the **name** of each **author along** with the **titles** of their **books**

```
[
    {
        "name": "Jane Austen",
        "books": [
            {
                "title": "Pride and Prejudice"
            },
            {
                "title": "Emma"
            }
        ]
    },
    {
        "name": "Charlotte Brontë",
        "books": [
            {
                "title": "Jane Eyre"
            }
        ]
    }
]
```

AuthorList GET Request - Example



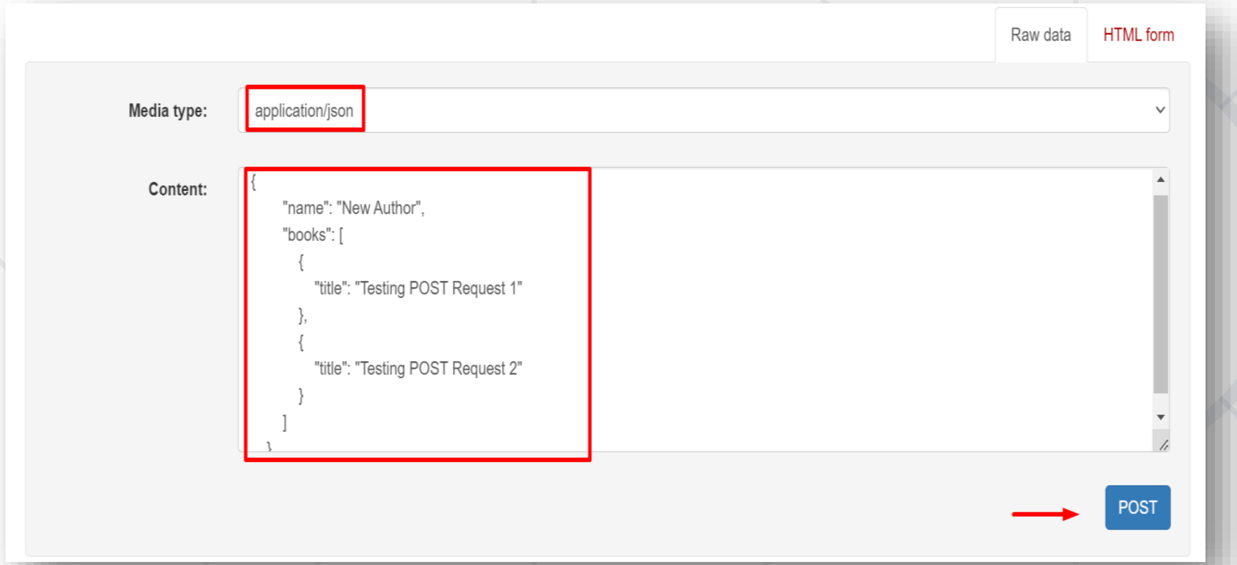
Author List

```
GET /api/authors/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "name": "Jane Austen",
        "books": [
            {
                "title": "Pride and Prejudice"
            },
            {
                "title": "Emma"
            }
        ]
    },
    {
        "name": "Charlotte Brontë",
        "books": [
            {
                "title": "Jane Eyre"
            }
        ]
    }
]
```

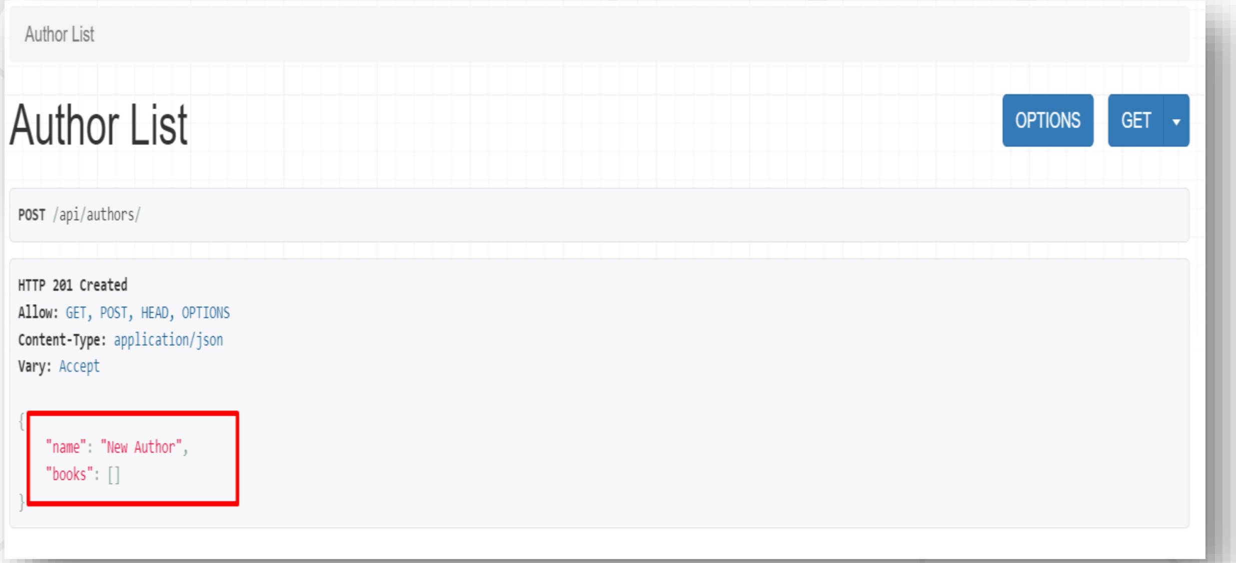
AuthorList POST Request - Example



Media type: application/json

Content:

```
{
    "name": "New Author",
    "books": [
        {
            "title": "Testing POST Request 1"
        },
        {
            "title": "Testing POST Request 2"
        }
    ]
}
```



Author List

POST /api/authors/

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

{
    "name": "New Author",
    "books": []
}
```

The author was successfully created but the books list is empty

Modifying the AuthorSerializer

- In your **AuthorSerializer**, you **need** to **override** the **create()** **method** to **handle** the **creation** of **related Book instances**

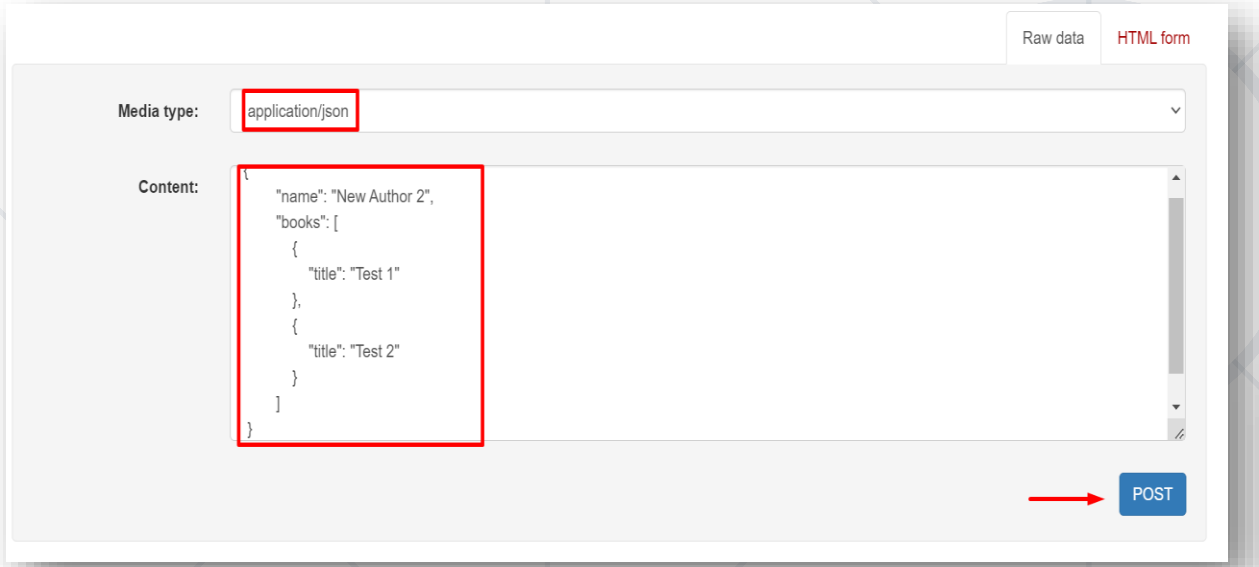
```
class AuthorSerializer(serializers.ModelSerializer):
    books = BookSerializer(many=True) # Remove read_only=True

    class Meta:
        model = Author
        fields = ['name', 'books']

    def create(self, validated_data):
        books_data = validated_data.pop('books')
        author = Author.objects.create(**validated_data)
        for book_data in books_data:
            Book.objects.create(author=author, **book_data)
        return author
```

The method takes care of creating both the Author instance and the related Book instances when a POST request with nested data is received

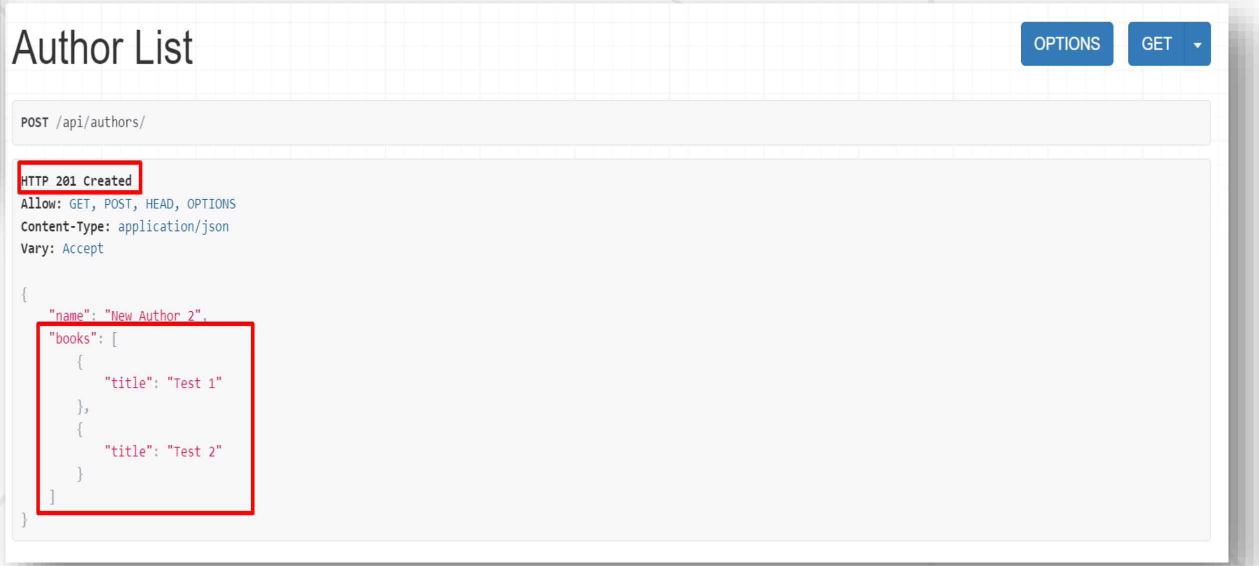
AuthorList POST Request - Example



Media type: application/json

Content:

```
{
    "name": "New Author 2",
    "books": [
        {
            "title": "Test 1"
        },
        {
            "title": "Test 2"
        }
    ]
}
```



```
POST /api/authors/

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

{
    "name": "New Author 2",
    "books": [
        {
            "title": "Test 1"
        },
        {
            "title": "Test 2"
        }
    ]
}
```

The author and the related books were successfully created

Benefits of Generic API Views

- The **generic views** **save** you **time** and **effort** by providing **commonly used functionalities** **out of the box**
- You can **customize** their **behavior** by **overriding** **methods** or **attributes** as needed
- Additionally, they **work** **seamlessly** with DRF **serializers** and Django **models** to provide a **consistent** and **efficient** way to build **RESTful APIs**

Live Demo

Generic API Views - CRUD Operations

Authentication and Permissions

Authentication and Permissions

- Authentication and permissions are **essential** components for **controlling** access to your APIs
- Authentication refers to the process of verifying the identity of a **user** or **system** requesting your API
- Permissions determine whether a **requester** is allowed to perform a **specific** action on a **particular** **resource** within your API

Built-in Authentication Classes

- DRF provides various **built-in authentication classes**
  - **Token Authentication**
    - Uses a **token**, typically **generated** upon **successful login**, which is then **included** in subsequent **requests** to **authenticate** the **user**
  - **Session Authentication**
    - Relies on Django's **built-in session framework** and uses **session cookies** to **authenticate users** for each request

Built-in Authentication Classes

- **Basic Authentication**
  - Requires users to **include** their **credentials** in the **request headers**, which are then **base64-encoded** before transmission, providing a **simple authentication mechanism**
- **JWT Authentication**
  - Utilizes **JSON Web Tokens** (**JWT**) as a **secure way** to transmit information between parties as a **JSON object**, often used for **stateless authentication** in APIs

TokenAuthentication - Example

```
from rest_framework.authtoken.models import Token
from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework.authentication import TokenAuthentication

class LoginView(APIView):
    authentication_classes = [TokenAuthentication]

    def post(self, request):
        user = authenticate(username=request.data['username'],
password=request.data['password'])
        if user:
            token, created = Token.objects.get_or_create(user=user)
            return Response({'token': token.key})
        else:
            return Response({'error': 'Invalid credentials'}, status=401)
```

The TokenAuthentication class is utilized to authenticate requests using tokens

The Token model is used to generate and manage tokens

TokenAuthentication - Example

- Add **Token-Based Authentication** to your project
- Configure the **authentication classes** by adding '`rest_framework.authentication.TokenAuthentication`' to the `DEFAULT_AUTHENTICATION_CLASSES` in `settings.py`

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
}
```

Permission Classes

- DRF provides a range of **permission classes**, such as
  - **`IsAuthenticated`**
  - **`AllowAny`**
  - **`IsAdminUser`**
- DRF also allows you to define **custom permission classes** to fit your **specific requirements**

IsAdminUser - Simple Example

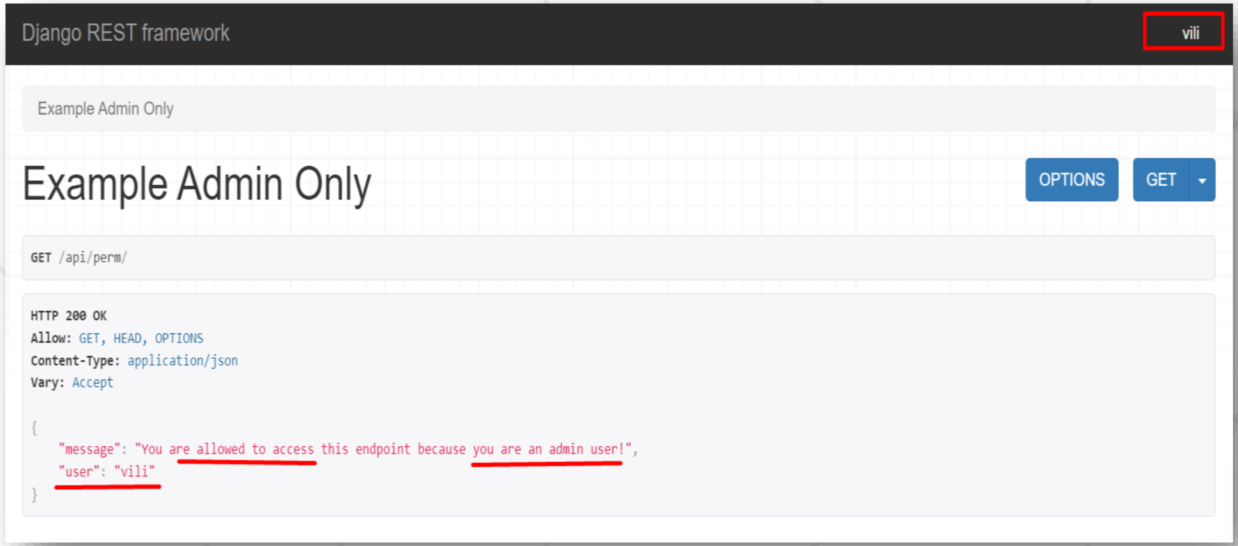
```
from rest_framework.permissions import IsAdminUser
from rest_framework.views import APIView
from rest_framework.response import Response

class ExampleAdminOnlyView(APIView):
    permission_classes = [IsAdminUser]

    def get(self, request, format=None):
        content = {
            'message': 'You are allowed to access this endpoint because you are
an admin user!',
            'user': str(request.user),  # Assuming request.user holds the
authenticated user object
        }
        return Response(content)
```

Only admins can access the endpoint

IsAdminUser - Simple Example



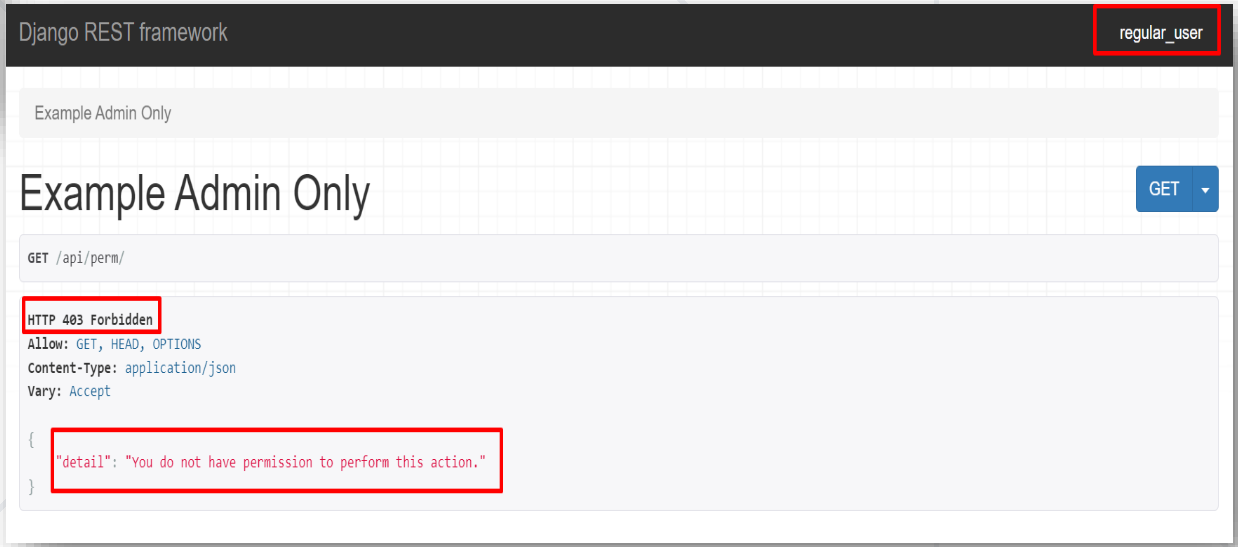
Example Admin Only

```
GET /api/perm/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "message": "You are allowed to access this endpoint because you are an admin user!",
    "user": "vili"
}
```

An admin user



Example Admin Only

```
GET /api/perm/
```

```
HTTP 403 Forbidden
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "You do not have permission to perform this action."
}
```

A regular user

Exception Handling in DRF

Exception Handling

- **Exception handling** in DRF views is essential for **managing errors** and providing **meaningful responses** to **clients**
- DRF's views are **equipped** to **handle various types of exceptions**, ensuring that **appropriate error responses** are **returned** to the **client**

Exceptions in DRF

- When working with DRF views, the following **types** of **exceptions** are commonly **handled**:
  - **Subclasses** of **`APIException`** raised within DRF
  - Django's **`Http404`** exception, indicating that the **requested resource** was **not found**
  - Django's **`PermissionDenied`** exception, signaling that the user **lacks** the **necessary permissions** to **access** a resource

Exceptions in DRF

- In each of these cases, DRF **automatically generates** a **response**
  - with an **appropriate** **HTTP status code** and **content-type**
- Additionally, the **response body** contains
  - **detailed information** about the **encountered error**, aiding in debugging and troubleshooting
- Most **error responses** **include** a **key** '`detail`' in the **body** of the **response**

Error Response - Example

```
DELETE http://api.example.com/foo/bar HTTP/1.1
Accept: application/json
```

Requesting a forbidden action

```
HTTP/1.1 405 Method Not Allowed
Content-Type: application/json
Content-Length: 42

{"detail": "Method 'DELETE' not allowed."}
```

Receiving an error response indicating that the DELETE method is not allowed on that resource

A key 'detail'

Validation Errors

- **Validation errors** are handled slightly differently, and **include** the **field names** as the **keys** in the **response**
- If the **validation error** is **not specific** to a **particular field**, then it uses the "non_field_errors" **key**, or whatever **string value** has been set for the NON_FIELD_ERRORS_KEY **setting**

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 94

{"amount": ["A valid integer is required."], "description":
["This field may not be blank."]}
```

Custom Exception Handling - Example

- In order to **alter** the style of the **response**, you could write a **custom exception handler**

```
from rest_framework.views import exception_handler

def custom_exception_handler(exc, context):
    # Call REST framework's default exception handler first, to get 
the standard error response
    response = exception_handler(exc, context)

    # Add the HTTP status code to the response data
    if response is not None:
        response.data['status_code'] = response.status_code

    return response
```

Custom Exception Handling - Example

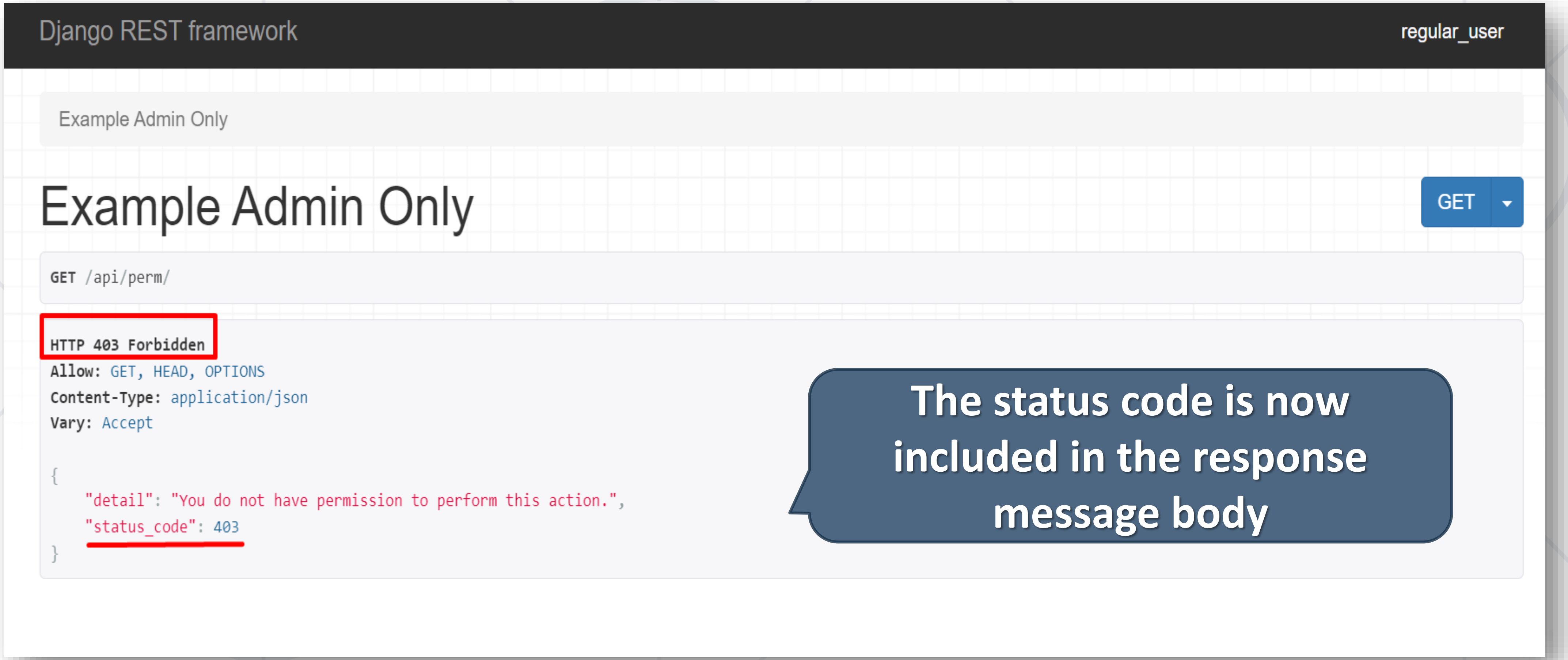
- The **custom exception handler** must also be **configured** in your settings, using the EXCEPTION_HANDLER setting **key**

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER':
'my_project.my_app.utils.custom_exception_handler'
}
```

- If **not specified**, the '**EXCEPTION_HANDLER**' setting **defaults** to the **standard exception handler** provided by REST framework

```
REST_FRAMEWORK = {
    'EXCEPTION_HANDLER': 'rest_framework.views.exception_handler'
}
```

Custom Exception Handling - Example



Example Admin Only

```
GET /api/perm/
```

```
HTTP 403 Forbidden
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "You do not have permission to perform this action.",
    "status_code": 403
}
```

The status code is now included in the response message body

APIException

- The **base class** for all **exceptions** raised **inside** an `APIView` **class** or `@api_view`
- To provide a **custom exception**
  - **subclass** `APIException`
  - **set** the `status_code`, `default_detail`, and `default_code` **attributes** on the **class**

Custom APIException - Example

- If your API **relies** on a **third-party** service that may sometimes be **unreachable**, you might want to **implement** an **exception** for the "**503 Service Unavailable**" **HTTP response code**

```
from rest_framework.exceptions import APIException

class ServiceUnavailable(APIException):
    status_code = 503
    default_detail = 'Service temporarily unavailable, try again later.'
    default_code = 'service_unavailable'
```

Summary

- Advanced **Serialization**
  - **Nested** Serializers
- **Generic Views** in DRF
- **Authentication** and **Permissions** in DRF
- **Exception Handling** in DRF

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

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