Unit_05 – RESTful APIs with Django Rest Framework

REST architecture

- REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other.
- REST-compliant systems, often called RESTful systems, are characterized by how they are stateless and separate the concerns of client and server.
- REST is not a standard but a set of architectural principles that guide the design of web services. Representational State Transfer is known for its simplistic nature, and utilizes interactions in order to communicate via HTTP protocols.

A Restful system consists of a:

- client who requests for the resources.
- server who has the resources.

- It is important to create REST API according to industry standards which results in ease of development and increase client adoption.

Architectural Constraints of RESTful API: There are six architectural constraints which makes any web service are listed below:

- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand

- **Uniform Interface:** It is a key constraint that differentiate between a REST API and Non-REST API. It suggests that there should be an uniform way of interacting with a given server irrespective of device or type of application (website, mobile app).

There are four guidelines principle of Uniform Interface are:

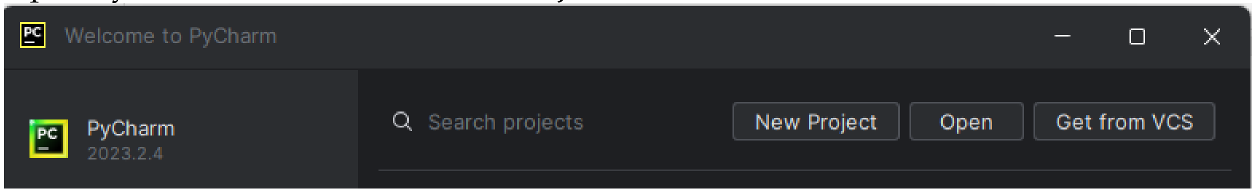
- **Resource-Based:** Individual resources are identified in requests. For example: API/users.
- **Manipulation of Resources Through Representations:** Client has representation of resource and it contains enough information to modify or delete the resource on the server, provided it has permission to do so. Example: Usually user get a user id when user request for a list of users and then use that id to delete or modify that particular user.
- **Self-descriptive Messages:** Each message includes enough information to describe how to process the message so that server can easily analyses the request.

- **Hypermedia as the Engine of Application State (HATEOAS):** It need to include links for each response so that client can discover other resources easily.

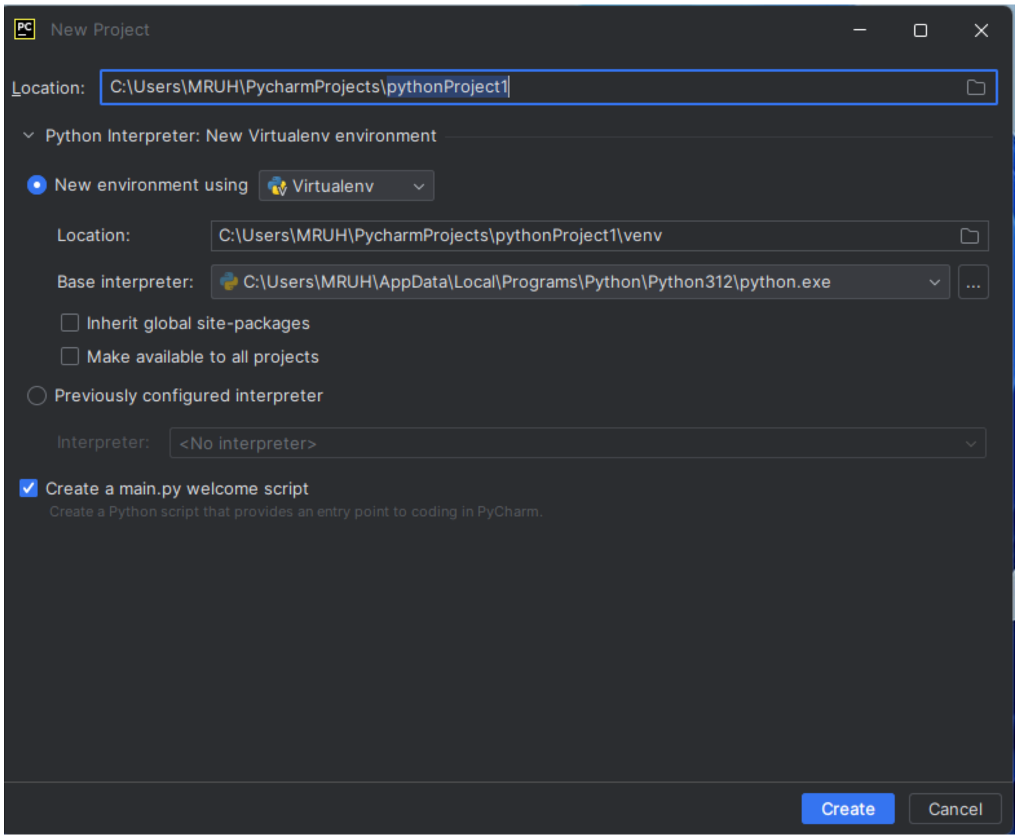
- **Statelessness:** A RESTful API should be stateless. In simple words, it means that it doesn't store any information about the user's session. Therefore, every single request should provide complete data to process it. Thus, it leads to greater availability of the API.
- **Cacheability:** The server's response should provide information on whether it should be cached and for what time or not. Caching the data that updates rarely improves performance and eliminates redundant client-server interactions.
- **Client-server architecture:** The client uses URIs to obtain resources. It doesn't concern how the server process the request. On the other hand, the server process and returns the resources. It doesn't impact a user interface in any way. Both client and server don't need to know about other responsibilities. Thus, they can evolve independently. It allows using a single API in many different clients, e.g., web browsers, mobile apps.
- **Layered system:** A REST API can consist of multiple layers, eg., business logic, presentation, data access. Moreover, layers shouldn't directly impact others. Further, the client shouldn't know if it's connected directly to the end server or intermediary. Therefore, we can easily scale the system or provide additional layers such as gateways, proxies, load balancers.
- **Code on demand:** This one is an optional constraint. The server can return a part of the code itself instead of the data in JSON format. The point is to provide specific operations on the data that the client can use directly. Although, it's not a common practice.

Setting up Django REST Framework

- Django is a Python web framework, thus requiring Python to be installed on machine.
- We can download python from the following website: https://www.python.org/
- Download the executable installer and run it. Check the boxes next to "Install launcher for all users (recommended)" then click "Install Now".
- Open PyCharm and Click on **New Project**.



- Select Directory and then give a name to our project and then Click on **Create**.



Location: C:\Users\MRUH\PycharmProjects\pythonProject1

Python Interpreter: New Virtualenv environment

New environment using: Virtualenv

Location: C:\Users\MRUH\PycharmProjects\pythonProject1\venv

Base interpreter: C:\Users\MRUH\AppData\Local\Programs\Python\Python312\python.exe

☐ Inherit global site-packages

☐ Make available to all projects

○ Previously configured interpreter

Interpreter: <No interpreter>

☑ Create a main.py welcome script

Create a Python script that provides an entry point to coding in PyCharm.

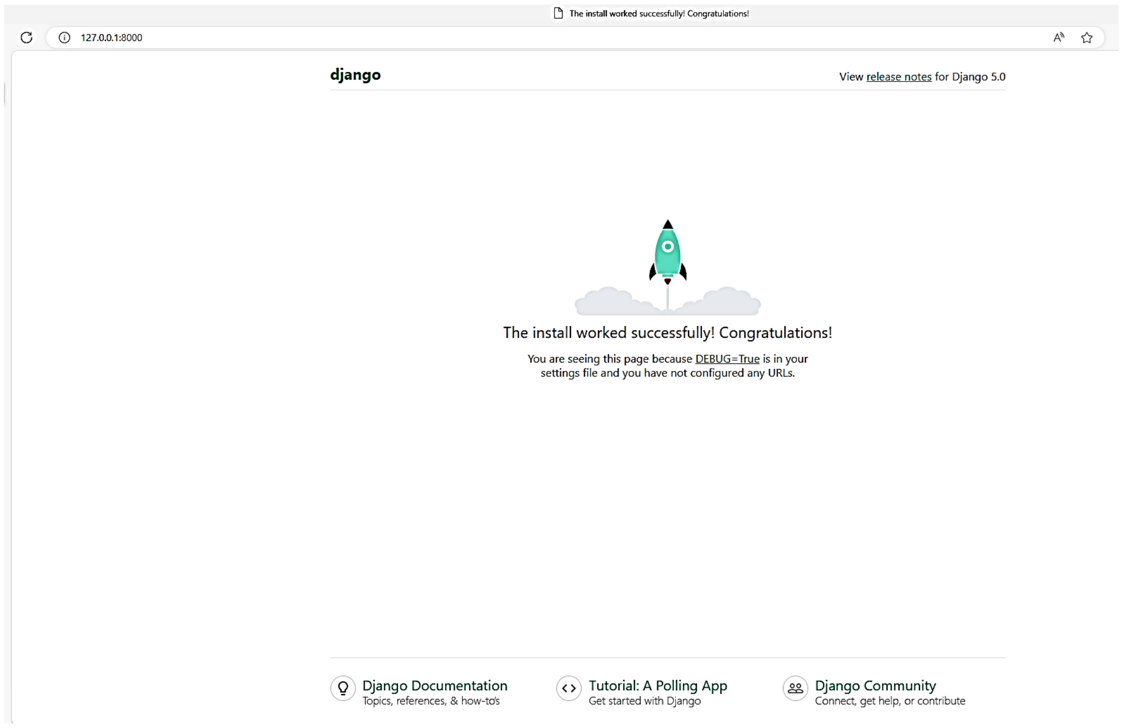
- Open the terminal in pycharm and type the following command to install Django
  ```
  pip install django
  ```
  This command fetches the django package from the Python Package Index (PyPI) using pip. After the installation has completed, you can pin your dependencies to make sure that you're keeping track of which Django version you installed:
- Type the following Command in pycharm Terminal to check the Django version.
  ```
  python -m django –version
  ```
- Type the following Command in pycharm Terminal to install Django REST Framework.
  ```
  pip install djangorestframework
  ```

- Start a project by using following command-
  ```
  django-admin startproject Myproject
  ```

- Change directory to Myproject.
  ```
  cd Myproject
  ```

- Start the server by typing the command
  ```
  python manage.py runserver
  ```

  By clicking the link http://127.0.0.1:8000/ we can see the result.


The install worked successfully! Congratulations!

127.0.0.1:8000

django

View release notes for Django 5.0

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to's

Tutorial: A Polling App
Get started with Django

Django Community
Connect, get help, or contribute

- To run the application in django rest framework after creating the project Add 'rest_framework' to your INSTALLED_APPS setting.

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

Build Web API using Django REST

➢ **Steps to Create a REST API Using Django REST Framework**

Step 1: Open PyCharm and Click on Create New Project.
Step 2: Select Directory and then give a name to our project and then Click on Create.
Step 3: Open the terminal in pycharm and type the following command to install Django

```
pip install django
```

Step 4: Type the following Command in pycharm Terminal to install Django REST Framework.

```
pip install djangorestframework
```

Step 5: Start a project by using following command-

```
django-admin startproject Myproject
```

Then change directory to Myproject.
```
cd Myproject
```

Step 6: Start the application by using following command-
```
python manage.py startapp myapp
```

Step 7: Add 'rest_framework' and 'application name (myapp) to your INSTALLED_APPS setting (in setting.py).
```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'mpapp',
]
```
Register the app URLs of "myapp" in the urls.py file:

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

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

Creating a REST API View

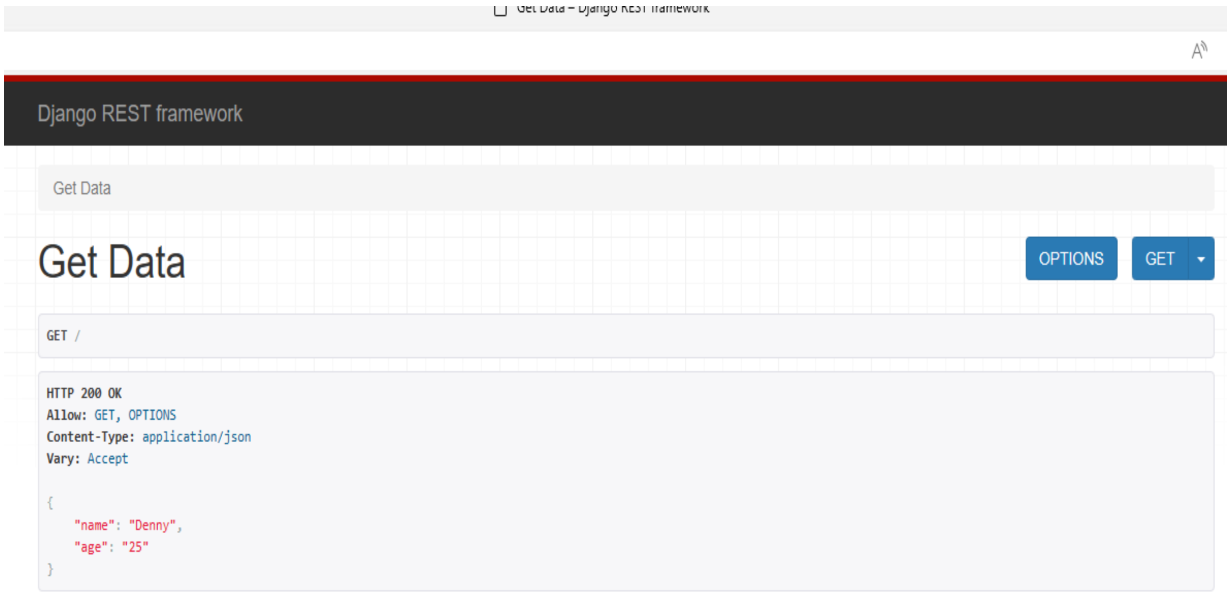
- In order to prevent errors, add a dummy view to the views.py file of the app. From the Django REST framework, you first have to import the @apiview decorator and Response object.
- This is because @apiview displays the API while Response returns sterilized data in JSON format.

```
@api_view(['GET'])
def getData(request):
    person = {'name':'Denny','age':'25'}
    return Response(person)
```

Building a URL Path for the App

```
from django.urls import path
from . import views
from django.conf import settings

urlpatterns = [
    path('', views.getData),
    path('add/', views.addItem),
    ]
```



```
GET /

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

{
    "name": "Denny",
    "age": "25"
}
```

RESTful API

- A RESTful API is an architectural style for an application program interface (API) that uses HTTP requests to access and use data.
- That data can be used to GET, PUT, POST and DELETE data types, which refers to the reading, updating, creating and deleting of operations concerning resources.
- An API for a website is code that allows two software programs to communicate with each other. The API spells out the proper way for a developer to write a program requesting services from an operating system or other application.
- A RESTful API -- also referred to as a RESTful web service or REST API -- is based on representational state transfer (REST), which is an architectural style and approach to communications often used in web services development.
- REST technology is generally preferred over other similar technologies.
- This tends to be the case because REST uses less bandwidth, making it more suitable for efficient internet usage.
- RESTful APIs can also be built with programming languages such as JavaScript or Python.

How RESTful APIs work

- A RESTful API breaks down a transaction to create a series of small modules.
- Each module addresses an underlying part of the transaction.

- This modularity provides developers with a lot of flexibility, but it can be challenging for developers to design their REST API from scratch.
- To understand how REST APIs work, it is critical to understand resources. A resource can be any information that could be named, such as a document or image, a collection of other resources, a non-virtual object, and more. Meanwhile, REST uses a resource identifier to recognize the specific resource involved in an interaction between components.
- The method is the type of request you send to the server. The four main resource methods that are associated with REST APIs are:
    - GET: This method allows for the server to find the data you requested and sends it back to you.
    - PUT: If you perform the 'PUT' request, then the server will update an entry in the database.
    - POST: This method permits the server to create a new entry in the database.
    - DELETE: This method allows the server to delete an entry in the database.

- **RESTful principles** - RESTful APIs adhere to constraints around having a uniform interface, being stateless, exposing directory structure-like URIs, and transferring XML, JSON or other representations of resources.
- **Resource Identification** - Every entity that can be manipulated or accessed in the API is represented as an HTTP resource exposed at a URI endpoint. URIs identify collections, elements, attributes etc.

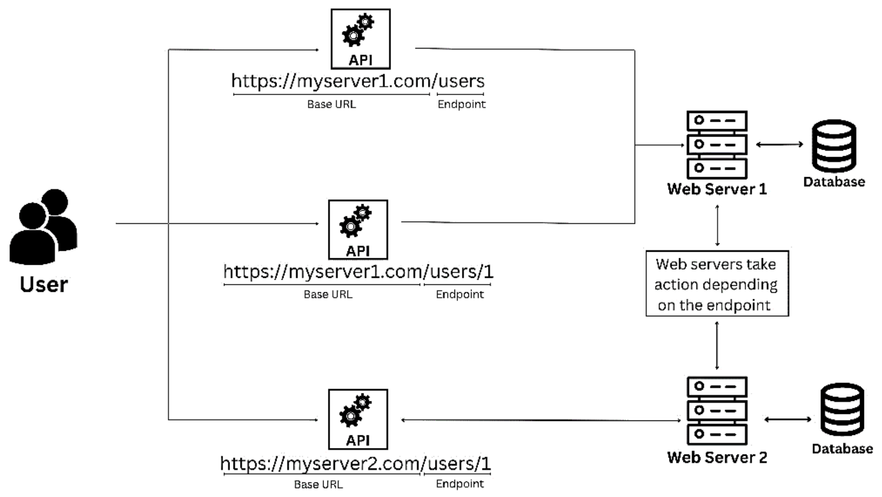
- **HTTP Methods** - REST uses standard HTTP methods to define interaction with resources. GET retrieves representations, POST creates resources, PUT updates them, DELETE deletes them.
- **Request Messages** - Clients send requests to REST API endpoints with HTTP headers containing metadata like authorization tokens, content-type, accept headers etc. Parameters help filter result sets or bind data.
- **Response Messages** - Servers return HTTP response codes indicating outcomes along with response headers describing the content. The message body contains representations of resources or data payloads.
- **Self-documentation** - REST APIs use HTTP conventions for response codes, verbs, and media types so APIs self-document how they are supposed to be used by clients.

API Endpoint

An endpoint is a component of an API, while an API is a set of rules that allow two applications to share resources. Endpoints are the locations of the resources, and the API uses endpoint URLs to retrieve the requested resources.

How does an API Endpoint work?

- An endpoint in the context of an API (Application Programming Interface) is a specific URL linking to a particular resource. When interacting with an API, endpoints can execute specific activities like requesting data or triggering a process.
- Consider an API for user management containing the following endpoints:


API
https://myserver1.com/users
Base URL
Endpoint
https://myserver1.com/users/1
https://myserver2.com/users/1
User
Web Server 1
Web Server 2
Database
Web servers take action depending on the endpoint

- In this diagram above, all the requests having the same base URL go to the same server.
- For example, https://myserver1.com requests go to "Web Server 1," and https://myserver2.com requests go to "Web Server 2".
- The webserver takes different actions based on the API endpoint. We use the /users endpoint to manage the collection of users as a whole and /users/{id} endpoint to manage individual users.
- Endpoints are an important concept in API design and are used to define the various resources and actions that can be performed through an API.

Endpoints with Multiple Protocols

- An endpoint is essentially a combination of a URL and an HTTP method.
- The HTTP method defines the type of operation supported by that endpoint. An endpoint can therefore support multiple protocols by allowing different HTTP methods.
- For instance, consider an endpoint /users representing a collection of users in an API. This endpoint could support the following protocols:
  - HTTP GET: Retrieves all the users from the collection using the HTTP GET technique
  - HTTP POST: We can add a new user to the collection using the HTTP POST technique
  - HTTP PUT: Update the full collection of users using the HTTP PUT technique (i.e., replacing it with a new collection)

  - HTTP DELETE: Delete one or more users from the collection using the HTTP DELETE technique
- Here, the same endpoint (/users) supports multiple protocols (GET, POST, PUT, DELETE) by allowing different HTTP methods. As a result, the API may offer a variety of functions for controlling the group of users from a single endpoint.

Practices for designing and developing API endpoints

- API endpoints are essential to the performance of any application. The following best practices can help to design, develop, and maintain endpoints that are reliable, scalable, user-friendly, and secure.

1. ***Create a predictable and intuitive API endpoint structure***

It's important to use a clear and intuitive naming convention and structure when defining your API's endpoints. When possible, the same conventions should apply to every API in your digital portfolio.

2. ***Implement secure authentication mechanisms***

API endpoints are the doorways to an application's data, which makes them appealing attack targets. API authentication involves verifying the identity of a client that is making an API request, and it plays a crucial role in strengthening an API's security posture. It's therefore important to leverage well-established authentication mechanisms especially if the endpoints provide access to sensitive data.

3. ***Validate and sanitize input data***

Input validation is the process of confirming that any data that's sent to the API follows the expected format and constraints, while sanitization helps ensure that input data does not include harmful characters in order to prevent any malicious code.

4. ***Clearly document every API endpoint***

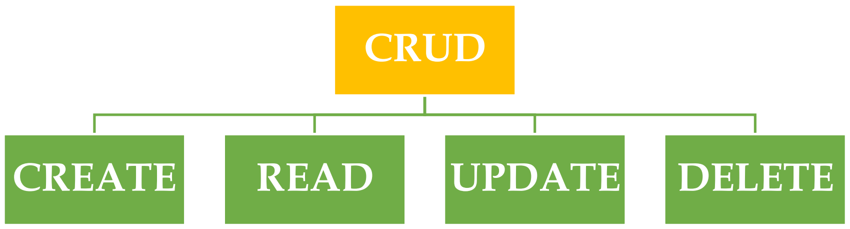
Private API documentation facilitates cross-team collaboration and reduces code duplication, while public API documentation helps potential consumers understand and experiment with an API.

5. ***Continually test and monitor your API endpoints***

API testing helps ensure that an API's endpoints work as expected—even as the API evolves. Unit tests confirm that a single endpoint returns the correct response to a given request, while end-to-end tests validate complex workflows that may involve multiple endpoints. API test automation can help team's automatically surface issues throughout the development process.

CRUD operation with Django REST Framework

- CRUD is the acronym for ***CREATE, READ, UPDATE and DELETE***.
- These terms describe the four essential operations for creating and managing persistent data elements, mainly in relational and NoSQL databases.
- Applications that use persistent storage—those that retain data even after the machine is powered down—perform CRUD operations.



- **CREATE:**
  - CREATE operation adds a new record to a database.
  - Create view will use the POST method for inserting data into our database.

```
@api_view(['POST'])
def add_items(request):
    item = ItemSerializer(data=request.data)

    if Item.objects.filter(**request.data).exists():
    raise serializers.ValidationError('This data already
exists')

    if item.is_valid():
        item.save()
        return Response(item.data)
    else:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

- **READ:**
  - Returns records (or documents or items) from a database table (or collection or bucket) based on some search criteria.
  - This view_items function will either show all the data or filtered data queried by the user according to the category, subcategory, or name.

```
@api_view(['GET'])
def view_items(request):

    if request.query_params:
        items =
Item.objects.filter(**request.query_params.dict())
    else:
        items = Item.objects.all()

    if items:
        serializer = ItemSerializer(items, many=True)
        return Response(serializer.data)
    else:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

- **UPDATE:**
  - Used to modify existing records in the database.
  - Update view uses the POST method to update a particular item from the database

```
@api_view(['POST'])
def update_items(request, pk):
    item = Item.objects.get(pk=pk)
    data = ItemSerializer(instance=item, data=request.data)

    if data.is_valid():
        data.save()
        return Response(data.data)
    else:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

- **DELETE:**
  - Allow the user to remove records from the database.
  - Delete view function we will use the DELETE method to remove the items.

```
@api_view(['DELETE'])
def delete_items(request, pk):
    item = get_object_or_404(Item, pk=pk)
    item.delete()
    return Response(status=status.HTTP_202_ACCEPTED)
```

Authentication in Django REST Framework

- Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with. The permission and throttling policies can then use those credentials to determine if the request should be permitted.
- REST framework provides several authentication schemes out of the box, and also allows you to implement custom schemes.
- Authentication always runs at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed.
- The `request.user` property will typically be set to an instance of the `contrib.auth` package's `User` class.

- The `request.auth` property is used for any additional authentication information, for example, it may be used to represent an authentication token that the request was signed with.

How authentication is determined

- The authentication schemes are always defined as a list of classes. REST framework will attempt to authenticate with each class in the list, and will set `request.user` and `request.auth` using the return value of the first class that successfully authenticates.
- If no class authenticates, `request.user` will be set to an instance of `django.contrib.auth.models.AnonymousUser`, and `request.auth` will be set to `None`.
- The value of `request.user` and `request.auth` for unauthenticated requests can be modified using the `UNAUTHENTICATED_USER` and `UNAUTHENTICATED_TOKEN` settings.

➢ **Types of authentication**

BasicAuthentication

This authentication scheme uses HTTP Basic Authentication, signed against a user's username and password. Basic authentication is generally only appropriate for testing.

If successfully authenticated, `BasicAuthentication` provides the following credentials.

- `request.user` will be a Django `User` instance.
- `request.auth` will be `None`.

Unauthenticated responses that are denied permission will result in an `HTTP 401 Unauthorized` response with an appropriate WWW-Authenticate header. For example:

```
WWW-Authenticate: Basic realm="api"
```

TokenAuthentication

- This authentication scheme uses a simple token-based HTTP Authentication scheme. Token authentication is appropriate for client-server setups, such as native desktop and mobile clients.
- To use the `TokenAuthentication` scheme you'll need to configure the authentication classes to include `TokenAuthentication`, and additionally include `rest_framework.authtoken` in your `INSTALLED_APPS` setting.

SessionAuthentication

- This authentication scheme uses Django's default session backend for authentication. Session authentication is appropriate for AJAX clients that are running in the same session context as your website.
- If successfully authenticated, `SessionAuthentication` provides the following credentials.

  - `request.user` will be a Django `User` instance.
  - `request.auth` will be `None`.
- Unauthenticated responses that are denied permission will result in an `HTTP 403 Forbidden` response.

RemoteUserAuthentication

- This authentication scheme allows you to delegate authentication to your web server, which sets the `REMOTE_USER` environment variable.
- To use it, you must have `django.contrib.auth.backends.RemoteUserBackend` (or a subclass) in your `AUTHENTICATION_BACKENDS` setting. By default, `RemoteUserBackend` creates `User` objects for usernames that don't already exist.

API View to handle Requests and Responses

➢ **Request**

- REST framework's Request class extends the standard HttpRequest, adding support for REST framework's flexible request parsing and request authentication.

➢ **Request parsing**

- REST framework's Request objects provide flexible request parsing that allows you to treat requests with JSON data or other media types in the same way that you would normally deal with form data.

✓ **.data**

- `request.data` returns the parsed content of the request body. This is similar to the standard `request.POST` and `request.FILES` attributes except that:
- It includes all parsed content, including *file and non-file* inputs.
- It supports parsing the content of HTTP methods other than `POST`, meaning that you can access the content of `PUT` and `PATCH` requests.
- It supports REST framework's flexible request parsing, rather than just supporting form data. For example you can handle incoming JSON data similarly to how you handle incoming form data.

✓ **.query_params**

- `request.query_params` is a more correctly named synonym for `request.GET`.
- For clarity inside your code, we recommend using `request.query_params` instead of the Django's standard `request.GET`. Doing so will help keep your codebase more correct and obvious - any HTTP method type may include query parameters, not just `GET` requests.

- ✓ **.parsers**
  - The `APIView` class or `@api_view` decorator will ensure that this property is automatically set to a list of `Parser` instances, based on the `parser_classes` set on the view or based on the `DEFAULT_PARSER_CLASSES` setting.

➢ Response

- REST framework supports HTTP content negotiation by providing a `Response` class which allows you to return content that can be rendered into multiple content types, depending on the client request.
- The `Response` class subclasses Django's `SimpleTemplateResponse`. `Response` objects are initialised with data, which should consist of native Python primitives. REST framework then uses standard HTTP content negotiation to determine how it should render the final response content.

✓ **Creating responses with Response ()**

- **Signature:** `Response(data, status=None, template_name=None, headers=None, content_type=None)`
- Unlike regular `HttpResponse` objects, you do not instantiate `Response` objects with rendered content. Instead you pass in unrendered data, which may consist of any Python primitives.
- The renderers used by the `Response` class cannot natively handle complex datatypes such as Django model instances, so you need to serialize the data into primitive datatypes before creating the `Response` object.
- You can use REST framework's `Serializer` classes to perform this data serialization, or use your own custom serialization.
- Arguments:
  - `data`: The serialized data for the response.
  - `status`: A status code for the response. Defaults to 200. See also status codes.
  - `template_name`: A template name to use if `HTMLRenderer` is selected.
  - `headers`: A dictionary of HTTP headers to use in the response.
  - `content_type`: The content type of the response. Typically, this will be set automatically by the renderer as determined by content negotiation, but there may be some cases where you need to specify the content type explicitly.

Review Questions

1. Illustrate the REST architecture in detail.
2. Summarize the procedure to setting up Django REST Framework.
3. How to Build Web API using Django REST?
4. What is an API Endpoint? How to create An API Endpoint with Django REST Framework?
5. Define and explain RESTful API.
6. Explain in detail CRUD operation with Django REST Framework.
7. List and describe different User Authentication in Django REST Framework (DRF).
8. Show how to use API View to handle Requests and Responses.